KLAVA: a Java package for distributed and mobile applications. Reference manual Version 2

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Viale Morgagni 65, 50134 Firenze, Italy
http://www.lorenzobettini.it

March 23, 2011

Contents

1 The Java package KLAVA **3**

2 KLAVA: Basic Concepts and Architecture **3**
2.1 Tuples . . . 4
2.2 Tuple Spaces . . . 6
2.3 Localities . . . 8
2.4 Nodes . . . 8
2.4.1 Node functionalities . . . 9
2.4.2 Node connectivity . . . 10
2.4.3 Locality Resolution . . . 11
2.5 Processes & Node Coordinators . . . 12
2.6 Some examples . . . 15
2.7 Code mobility in KLAVA . . . 19

3 Programming Graphical Applications **23**

4 Three Example Applications **27**
4.1 A news gatherer . . . 27
4.2 Load balancing . . . 30
4.3 A chat system . . . 35

References **44**

1 The Java package KLAVA

In this document we illustrate the framework KLAVA, a Java package for implementing distributed applications that can exploit mobile code and run over a heterogeneous network environment. KLAVA is based on the KLAIM coordination paradigm with multiple distributed tuple spaces (*Kernel Language for Agent Interaction and Mobility*) [De Nicola *et al.*, 1998; Bettini *et al.*, 2003]. Thus, the underlying model enables *space uncoupling*, *time uncoupling* and *destination uncoupling*, and *asynchronous*, *associative* and *anonymous* communication. This programming model is suitable for distributed applications, mobile agents, and, more in general, mobile code (for an overview of communication models for mobile agent systems we refer to [Deugo, 2001]). KLAVA also provides support for moving processes and all the code they will need for execution at remote sites.

KLAVA is also used as the run-time system for the programming language X-KLAIM ([Bettini, 2003b; Bettini, 2003a; Bettini & De Nicola, 2005]) in that the X-KLAIM compiler generates Java code that relies on the KLAVA framework. Thus, X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm.

The framework was originally developed in [Bettini, 1998], and presented in details in [Bettini *et al.*, 2002b]; however, the KLAVA framework we present here differs from the previous presentations in that it relies on the hierarchical model of KLAIM introduced in [Bettini *et al.*, 2002a]. Many parts of the framework were almost complitely written from scratch (with respect to version 1) and probably old applications are not compatible with this new version. This new implementation relies on the IMC (*Implementing Mobile Code*) framework [Bettini *et al.*, 2005]. The use of IMC simplified most of the re-implementation of KLAVA and introduced many flexible features.

2 KLAVA: Basic Concepts and Architecture

In this section, we present the classes of the package `klava`[1]. Some of `klava` classes can already be used as they are (e.g., the class `Tuple`), while others have to be specialized through inheritance and method overriding (e.g., the class `KlavaProcess`).

Before describing KLAVA we give a very brief introduction to KLAIM (we refer the interested reader to [Bettini *et al.*, 2003] and to the KLAIM web page, `http://music.dsi.unifi.it`, for more complete descriptions of the formal model).

KLAIM is based on the notion of *locality* and relies on a Linda-like communication model. Linda [Carriero & Gelernter, 1989b; Gelernter, 1985; Gelernter, 1989] is a coordination language with asynchronous communication and shared memory. The shared space is named *tuple space*, a multiset of *tuples*; These are containers of information items (called *fields*). There can be *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier.

Tuples are anonymous and content-addressable. *Pattern-matching* is used to select tuples in a tuple space: two tuples match if they have the same number of fields and corresponding fields match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, tuple $(\text{“foo”}, \text{“bar”}, 100 + 200)$ matches with $(\text{“foo”}, \text{“bar”}, !Val)$. After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities: *physical localities* are the identifiers through which nodes can be uniquely identified within a net; *logical localities* are symbolic names for nodes. A reserved logical locality, `self`, can be used by processes to refer to their execution node. Physical

[1] Notice that we changed the name of the package from `Klava` to `klava` since package names are advised to be lowercase.

localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can execute the following operations over tuple spaces and nodes:

- **in**(t)@l: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever a matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- **read**(t)@l: differs from **in**(t)@l only because the tuple t' selected by pattern-matching is not removed from the tuple space located at l.
- **out**(t)@l: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**(P)@l: spawns process P for execution at l.

In the original KLAIM model, and thus also in the original KLAVA framework, *nodes* were merely the execution engines for processes and they had to be part of a *net*. A node in a net was accessible by any other node in the same net, once its (physical) locality was known. This led to a *flat* model in that nodes could not contain other nodes, thus subnets and hierarchical nets could not be directly modeled. Finally, while the separation between concrete and symbolic addresses of nodes implemented a sort of dynamism, still this dynamism did not suit open networks well. Indeed, the model had a static flavor that leads to a sort of "closed world": apart from the creation of new nodes, the topology of KLAIM nets could not be changed.

This new version of KLAVA adopts the hierarchical model of KLAIM, presented in [Bettini *et al.*, 2002a; Bettini, 2003a]: mechanisms for dynamically updating nodes' allocation environments and for explicitly dealing with node connectivity are now supplied. Moreover, a new category of processes, called *NodeCoordinators* is available, which, in addition to the KLAIM operations, can execute coordination operations for establishing new connections, for accepting connection requests and for removing connections. Node connectivity features will be described in Section 2.4.

2.1 Tuples

Tuples are sequences of information items called *fields* and are the basic tools for data elaboration and information exchange. There are two kinds of tuple fields: *actual fields* (i.e., expressions, processes, localities, constants, initialized variables) and *formal fields* (i.e., non initialized variables).

The class `Tuple` includes methods for handling tuples, such as creating tuples, adding elements to a tuple, getting an element of a tuple, etc. A tuple can be created by passing a `Vector` object, containing all tuple elements, to the `Tuple` constructor, such as:

```
Vector v = new Vector() ;
v.addElement( o1 ) ;
v.addElement( o2 ) ;
v.addElement( o3 ) ;
Tuple t = new Tuple( v ) ;
```

or by first creating an empty tuple and then adding elements using the method `add(Object o)`. To make tuple construction easier, a limited number of overloaded constructors is available such as:

```
public Tuple( Object o1 )
public Tuple( Object o1, Object o2 ) ...
```

so that one can create a new tuple simply by writing:

```
Tuple t = new Tuple( o1, o2, o3 ) ;
```

Tuples are anonymous and content-addressable and *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type and two actual fields match only if they are identical (two formals never match).

After matching, the variable of a formal field will get the value of the field it has matched.

Pattern-matching is implemented through the method `match()` of the class `Tuple` that takes a tuple as parameter and checks the matching with the current tuple. The method `match()` also performs the binding of the formals, in case the matching succeeds.

The interface `TupleItem` can be used for handling tuple fields. Its declaration is as follows:

```
public interface TupleItem extends java.io.Serializable {
  public boolean isFormal() ; // is it a formal?
  public void setValue(Object o) throws KlavaException ; // for updating
  public boolean equals(Object o) ; // are they equal?
  public Object duplicate(); // duplicate an item
}
```

`TupleItem`'s methods are used by the matching algorithm. More specifically, `isFormal()` is used to test whether a tuple field is a formal, `setValue()` to update a formal field with an actual value, and `equals()` to test whether two actual fields match. As usual, the semantics of these methods must be specified by the classes that implement the interface. The package `klava` makes available some wrapper classes for standard data types that implement this interface: `KString`, `KInteger`, `KBoolean` and `KVector`.

It is assumed that a `TupleItem` created with the default constructor (i.e., with no parameters) is a formal. We think that this is better than having a method `setFormal`, which may cause inconsistencies among aliases. Below we provide an example:

Example 2.1. This example shows how to use tuples and `match()`:

```
KString s = new KString() ; // formal declaration
KInteger i = new KInteger() ; // formal declaration
Tuple t1 = new Tuple( s, i ) ;
Tuple t2 = new Tuple( new KString("Hello"), new KInteger(10) ) ;
t2.match( t1 ) ; // true
System.out.println( "s now is : " + s );
System.out.println( "i now is : " + i );
```

□

Notice that the values of formal fields are automatically updated (by means of the method `setValue()`).

Alternatively, a `Class` object can be used for expressing a formal field. In this case, the class of that specific tuple item, must have a constructor that accepts a `String` (that is the representation of the value for the tuple item). After a successful matching, the `Tuple` method `getItem()`, `Object getItem(int index)` (indexes start from 0), can be used to retrieve the value bound to a formal field.

Example 2.2. An alternative version of Example 2.1 is the following one:

```
Tuple t1 = new Tuple( String.class, Integer.class ) ;
Tuple t2 = new Tuple( new String("Hello"), new Integer(10) ) ;
t2.match( t1 ) ; // true
System.out.println( "matched string : " + t1.getItem(0) ) ;
System.out.println( "matched integer : " + t1.getItem(1) ) ;
```

□

The method `match()` relies on types, thus, e.g., a `KString` cannot match a `String`. Indeed, the KLAVA matching mechanism is object based, but not object-oriented, in the sense that subtyping is not considered while checking whether two tuple fields match; namely, the static types must be the same. Thus, the internal structure of objects is no further inspected and, in particular, only the method `equals()` is employed in order to test whether two actual fields match. There is only one exception that is needed to avoid that the matching mechanism limits the exchange of code and to make it possible to match a tuple containing processes without knowing their concrete classes in advance: actual processes, that are instances of subclasses of `KlavaProcess` (explained later), are retrieved by means of tuple fields of class `KlavaProcessVar` (see Section 2.4.1).

The KLAVA system automatically assigns a unique identifier to each tuple; such an identifier can be considered as a GUID (*Global Unique Identifier*); after the matching, the identifier of the matching tuple is stored in the template used for retrieving a tuple. This is useful when using the same template for retrieving another tuple: once a tuple has been retrieved through a template tuple, the method `resetOriginalTemplate()` can be called to reset the template, i.e., its formal fields are initialized to empty values again, and such template can be used for retrieving another tuple. Since the identifier of a tuple that has already been retrieved is stored in the template, the pattern-matching will consider only for tuples not already inspected. This provides an easy mechanism for iterating through a tuple space. Already retrieved tuples are discarded, during the matching, by the method `preMatch()`, that can also be redefined in case a finer filtering mechanism is needed.

Notice that by default a tuple stores all the tuples that it has matched; in cases when you remove a tuple from a tuple space, and then put it back this behavior will not make your code work; in this case you can use `setHandleRetrieved()` on the tuple specifying `false`: this will make the tuple not store the tuples that it has matched. An example of this is in Section 4.2.

Example 2.3. In the following example, after the first matching between `t2` and `t1` the template `t1` is reset; trying the matching with `t2` once again will fail, while it will succeed with another matching tuple, `t3`:

```
KString s = new KString() ; // formal declaration
KInteger i = new KInteger() ; // formal declaration
Tuple t1 = new Tuple( s, i ) ;
Tuple t2 = new Tuple( new KString("Hello"), new KInteger(10) ) ;
Tuple t3 = new Tuple( new KString("World"), new KInteger(20) ) ;
t2.match( t1 ) ; // true
t1.resetOriginalTemplate();
t2.match( t1 ) ; // false: already matched
t3.match( t1 ) ; // true
```

□

2.2 Tuple Spaces

Tuple spaces are multisets of tuples. The interface `TupleSpace` includes methods to place tuples in and retrieve tuples from a tuple space (the `EventGenerator` will be explained later).

```
01: public interface TupleSpace extends EventGenerator {
02:     public boolean in( Tuple t ) throws InterruptedException;
03:     public boolean read( Tuple t ) throws InterruptedException;
04:     public void out( Tuple t ) ;
05:
```

```
06:   public boolean read_t(Tuple t, long TimeOut) throws InterruptedException;
07:   public boolean in_t(Tuple t, long TimeOut) throws InterruptedException;
08:
09:   public boolean read_nb(Tuple t);
10:   public boolean in_nb(Tuple t);
11:
12:   public int length();
13:   public void removeTuple(int i);
14:   public void removeAllTuples();
15:   public Enumeration<Tuple> getTupleEnumeration();
16: }
17:
```

In particular:

- **in**(t): looks for a tuple t' that matches t. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- **read**(t): differs from **in**(t) only because the matching tuple t' is not removed from the tuple space.
- **out**(t): adds the tuple t to the tuple space.

Two other methods of this class that implement operations over tuple spaces are:

```
public boolean read_nb( Tuple t )
public boolean in_nb( Tuple t )
```

`read_nb` and `in_nb` act like `read` and `in`, but, if no matching tuple is found, they do not block the executing process and simply return `false`. Some versions of Linda also introduce such operations, called **readp** and **inp** [Carriero & Gelernter, 1989a]. These variants are useful when one wants to search for a matching tuple in a tuple space with no risk of blocking. For instance, `read_nb` can be used to test whether a tuple is present in a tuple space.

One can provide its own implementation of this interface. The package already provides an implementation that uses a `Vector` to store tuples: `TupleSpaceVector`.

Example 2.4. Below we provide an example that uses a `TupleSpace`:

```
TupleSpace TS = new TupleSpaceVector() ;
KString s = new KString() ; // formal
KInteger i = new KInteger() ; // formal
Tuple t = new Tuple( s, new KInteger(10) ) ;
TS.out( new Tuple(new KString("Hello"), new KInteger(10)));
TS.out( new Tuple( new KInteger(10) ) ) ;
TS.in( t ) ; // withdraws the first tuple
TS.read( new Tuple( i ) ) ; // reads the second one
```

□

Iteration on a tuple space can be easily implemented by using the same template tuple and the method `resetOriginalTemplate()`, as shown in Section 2.1.

Due to network latency and bandwidth, network communications can be quite slow, hence, retrieving information may require more time than one is willing to wait. Moreover, the absence of matching tuples could block a process executing an **in**/**read** operation. To put upper bounds to the waiting time, a *time-out*, expressed in milliseconds, can be used. Therefore, "timed" versions of the blocking methods are supplied:

```
public boolean read_t( Tuple t, long TimeOut )
public boolean in_t( Tuple t, long TimeOut )
```

2.3 Localities

Localities are the tools that processes can use for referring to nodes of the net. We distinguish between two kinds of localities:

- *physical localities* are identifiers through which nodes can be uniquely identified within a net;
- *logical localities* are symbolic names for nodes. A distinct logical locality, `self`, can be used by processes to refer to their execution node.

Intuitively, physical localities have an absolute meaning within the whole net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought of as aliases for network resources. The association between logical and physical localities is modeled via a function that we call *allocation environment*; each node has an allocation environment that solves the logical localities there used.

There are three classes that handle localities. The abstract class `Locality` is the base class and implements the interface `TupleItem`. The other two classes `LogicalLocality` and `PhysicalLocality` are derived from this base class. A variable that represents a locality should always be declared as a `Locality` so that polymorphism can be used extensively.

Physical localities are strings following the syntax of IMC session identifiers:

```
<connection protocol identifier> - <protocol specific address>
```

where the first part identifies the specific communication protocol used for the session (i.e., for the actual connection/communication protocol) and the remaining part has a shape that (possibly) depends on the first part. IMC provides communication features for TCP, UDP and local pipes. For TCP and UDP the session identifiers will have shapes such as `tcp-<IP>:<port>` and `udp-<IP>:<port>`, respectively, while for local pipes `pipe-<any identifier>`. The big advantage of IMC session management is that to switch from a communication procotol to another one, all you have to do is change the session identifier, and nothing else. Notice that, in `<IP>:<port>`, the IP part is actually a numeric IP and not a DNS name.

A `PhysicalLocality` is initialized through a string representing a valid session identifiers. In case of a malformed string a `KlavaMalformedPhyLocalityException` exception will be thrown.

Notice that in previous versions of KLAVA a node could have only one `PhysicalLocality` and this was the only means to be addressed in a network. In this new implementation, a node can own more than one `PhysicalLocality`. Indeed now a `PhysicalLocality` is a session identifier; since a node can have many active sessions, it has many ways of being addressed in a network: through the session identifiers (i.e., physical localities) of its active sessions.

2.4 Nodes

As hinted in Section 2, any node will now play a double role: it is a computational environment for processes and the container of a tuple space, and a *gateway*, or *server*[2], that can manage a subnet of other nodes (clients). Moreover, nodes can act both as clients (belonging to a specific subnet) and as servers (taking charge of, possibly private, subnets). Logical localities represent the names that client nodes can specify when entering the subnet of a server node, and allocation environments, that can be dynamically updated with such information, actually represent dynamic tables mapping logical names (possibly not known in advance) into physical addresses; these mappings are allowed to change during the evolution. The client-server relation among nodes smoothly leads to a hierarchical model, also because of the way logical names are "resolved": in order to find the mapping for a locality, allocation environments of nodes in this hierarchy are now inspected from the bottom upwards. This resembles name resolution within DNS servers.

[2]In the following the terms *gateway* and *server* will be used interchangeably.

2.4.1 Node functionalities

Nodes are the loci where tuples and processes reside; they are also the execution engines for KLAVA processes. The class `KlavaNode` contains a single tuple space and exports methods for accessing this tuple space. These methods take as parameters a tuple and the locality of the destination node; if the operation refers to the current execution site, it is simply redirected to the local tuple space, otherwise a message will be sent to the (possibly remote) destination node.

```
public void out( Tuple t, Locality l ) throws KlavaException
public void read( Tuple t, Locality l ) throws KlavaException
public void in( Tuple t, Locality l ) throws KlavaException
public boolean read_nb( Tuple t, Locality l ) throws KlavaException
public boolean in_nb( Tuple t, Locality l ) throws KlavaException
public boolean read_t( Tuple t, Locality l ) throws KlavaException
public boolean in_t( Tuple t, Locality l ) throws KlavaException
```

By using these methods, processes can explicitly address the tuple space where a given operation must be executed. For instance, $\mathbf{out}(t, l)$ means that the tuple t must be placed at the tuple space located at l. There are also versions without the `Locality` parameter, that act directly on the node's tuple space.

The class `KlavaNode` also includes the method

```
public void eval( KlavaProcess P, Locality loc ) throws KlavaException
```

that corresponds to the KLAIM operation $\mathbf{eval}(P, l)$ that spawns process P for execution at node l.

`KlavaNode` declares the field `self` (of class `LogicalLocality`) that can be used to refer to the execution environment. Moreover, each node has an *allocation environment*, namely a sort of partial function that maps logical localities into physical localities. The environment of a node can be modified with the methods `addToEnvironment()`, `removeLogical()` and `removePhysical()`.

If the destination locality is a logical one, it will be first translated into the corresponding physical locality (if this cannot be done a `KlavaLoggicalLocalityException` will be thrown. This translation is done by the method `getPhysical()`.

When the translation of a logical locality into a physical locality is performed by a process further issues arise. In particular, `self` may not refer to the current execution node (instead, for instance, it could refer to the original node the process is coming from). This is also the reason why we provide also versions of the above operations without the destination locality. We will tackle this issue later.

The major difference between $\mathbf{eval}(P, l)$ and $\mathbf{out}(P, l)$ is that $\mathbf{eval}(P, l)$ automatically starts the execution of the process P at the remote site l, while $\mathbf{out}(P, l)$ does not. Indeed, $\mathbf{out}(P, l)$ simply posts a tuple containing the process P at the tuple space of l. The process has to be explicitly retrieved from the tuple space by means of a `KlavaProcessVar` (e.g, by a process executing at l), and explicitly started, as in the following example:

```
KlavaProcessVar PV = new KlavaProcessVar(); // formal
myNode.in( new Tuple(PV), self );
myNode.eval( PV.klavaProcess, self );
```

Notice that, since **out** has a tuple as a parameter, many processes can be delivered to a remote site at once.

Another way of executing a process on a node is via the method `executeProcess()` that takes another process as argument. Differently from **eval**, this method waits for the other process to terminate before returning.

Another important method of the class `KlavaNode` is

public void close() **throws** IMCException

This will close all the connections (sessions), terminates all the processes and node coordinators currently running on the node, and invokes `close()` on each node created with a `newloc()`. As for terminating the processes, `close()` waits for the processes to terminate; if they do not terminate within few seconds, it will throw an exception. A process is terminated by calling its `close()` (that subclasses can override in order to perform specific closing procedures) and then by invoking the method `interrupt()` of the Java class `Thread`.

2.4.2 Node connectivity

Two nodes can communicate only if they are connected somehow: either directly or indirectly, belonging to the same KLAVA net. Indeed, a server node acts as a *gateway* in that it allows client nodes, belonging to its subnet and possibly executing on different computers, to communicate with each other.

`KlavaNode` also provides all the methods for implementing node connectivity. These have to be used when a (client) node wants to enter the subnet managed by another node. Each method for establishing such a relation has also the complementary action that has to be performed by the server to accept a node in its net. All the following methods, implementing *privileged actions*, can be executed only by *node coordinators*, i.e., special processes, with "super user" functionalities, described in Section 2.5.

boolean login(Locality loc) **throws** KlavaException;
boolean accept(PhysicalLocality remote) **throws** KlavaException;
boolean accept(PhysicalLocality local, PhysicalLocality remote) **throws** KlavaException;

The first method has to be performed at the client node, and succeeds if the server (whose locality is specified as the parameter of `login()`) executes an `accept()` action. The physical locality of the connected client node is stored in the physical locality that is argument of `accept()`. Both operations block the executing processes on each node, until the connection is established. `login()` returns **false** in case the connection is not accepted. The second variant of `accept()` allows you to specify the `PhysicalLocality` where the server waits for login requests. If you use the first variant, i.e., without specifying the local physical locality, then the physical locality stored in the `mainPhysicalLocality` field will be used. This must be set (with the corresponding set method) otherwise an exception will be raised.

No logical locality is involved in this kind of connection. These are instead used when the following (complementary) methods are invoked:

boolean subscribe(Locality ploc, LogicalLocality lloc) **throws** KlavaException;
boolean register(PhysicalLocality remote, LogicalLocality lloc) **throws** KlavaException;
boolean register(PhysicalLocality local, PhysicalLocality remote, LogicalLocality lloc)
throws KlavaException;

Once again, the former has to be executed by the client node and the latter by the server. With `subscribe()` the client specifies also the logical locality with which it wants to become part of the server's net. Of course the connection is refused in case this logical locality is already used by another already connected client. The complementary operation in the server is `register()`, similar to `accept()` but which also stores the `LogicalLocality` of the client.

A client node can disconnect from a server by using the following methods (depending on the method used for the connection):

boolean logout(Locality loc) **throws** KlavaException;
boolean unsubscribe(Locality loc, LogicalLocality myloc) **throws** KlavaException;

In turn, the server can use the methods:

void disconnected(PhysicalLocality loc) **throws** KlavaException;
void disconnected(PhysicalLocality loc, LogicalLocality lloc) **throws** KlavaException;

for catching the disconnection events above. These methods can also be used for detecting connection failures; both methods return the physical locality of the node that has disconnected, the second method also returns the logical locality the node had subscribed with. Please, notice that `logout()` and `unsubscribe()` do not need to syncrhonize with the `disconnected()` on the server: indeed the server is not required to execute `disconnected()`; it can do it only to intercept disconnections.

In this scenario communications among nodes belonging to the same subnet take place, through the gateway node. In case of firewalls or network restrictions the access to a remote node may be permitted only through a server. For instance, an applet can only open a network connection towards the computer it has been downloaded from. If on this computer there is a `KlavaNode` running that is willing to act as a gateway, the applet is still able to indirectly communicate with all the nodes and, possibly, with applets that are part of that net managed by that gateway. In this sense, a `KlavaNode` gateway allows nodes to communicate even if they belong to different restricted domains.

`newloc()` is a privileged action that allows you to log a node (possibily created from scratch) to the current node and has many variants:

PhysicalLocality newloc() **throws** KlavaException;
PhysicalLocality newloc(NodeCoordinator P) **throws** KlavaException;
PhysicalLocality newloc(String classname, NodeCoordinator P) **throws** KlavaException;
PhysicalLocality newloc(KlavaNode node) **throws** KlavaException;

The first variant of `newloc()` creates a new `KlavaNode` and logs it into the current node subnet. The `PhysicalLocality` returned is a fresh one. The two nodes will communicate through a local pipe and indeed if you print the returned locality it will contain the `pipe` prefix (see Section 2.3). The second variant also takes as argument a node coordinator to be installed in the newly created node; As explained in Section 2.5, this is the only way of installing a node coordinator on another node, since node coordinators, due to security reasons cannot be sent for evaluation to another node. The third variant also the name of the Java class to be used for instantiating the new node. The last variant takes an already created `KlavaNode` instance and simply logs it into this node (returning the brand new physical locality).

Finally, we have a method that closes all sessions involving a specific locality, `closeSessions()`:

public void closeSessions(PhysicalLocality physicalLocality);

this means that all the active sessions that involve the given locality (as a local or remote end) will be closed, and if there's a `KlavaNodeCoordinator` performing an `accept()` or `register()` on the given locality, it will terminate too. This is used the chat server presented in Section 4.3.

2.4.3 Locality Resolution

The hierarchical KLAIM model is also characterized by the way logical localities are resolved: in order to evaluate locality names, whenever s_1 is logged in s_2, if a locality cannot be resolved by just using the allocation environment of s_1, then the allocation environment of s_2 (and possibly that of nodes to which s_2 is logged in) is also inspected. Thus, in order to find the mapping for a locality, allocation environments of nodes in this hierarchy are inspected from the bottom upwards. In particular, when a node is not able to solve a logical locality, it basically sends a message to all the gateways it is connected to and waits for the answer; these gateways in turns

may have to rely on its own gateway if it cannot solve it itself. The whole operation fails when a node in the hierarchy is not able to solve a logical locality and it is not connected to any gateway.

In particular, in this implementation of KLAVA, we considered nets structured as *graphs* (and not simple trees as in [Bettini, 2003a]) just like the model described in [Bettini *et al.*, 2002a]. The issues "which way to choose for resolving a name, when a node is connected to more than one server?" has the answer "the first we find". A future enhancement of KLAVA could be that of providing the programmer with customizable policies for choosing the resolution path.

Gateways are essential for communication: apart from direct connections, two nodes are guaranteed to interact only if there exists a node that acts as gateway for both. In order to implement this in an efficient way, a gateway has to keep track of all the nodes in its subnet (recursively). Thus when a server node accepts another node in its own subnet, it propagates the physical locality of the new client to the servers it is logged to (if there is one). This propagation mechanism allows the hierarchy to be up-to-date; of course disconnections have to be propagated as well. Notice however that propagations always take place from the client to the server, and never the other way round.

Nodes not directly connected but belonging to a common subnet can communicate thanks to a mechanism of message forwarding:

- if a node has to send a message to a node that it's not in its own subnet it forwards the message to all the servers it is logged to;
- a node can receive a message that is not destined to the node itself and in that case, if it's in its subnet it forwards it to that node, otherwise, in turn it will forwards it to all the servers it is connected to.

Notice that nodes created/logged through `newloc()` (Section 2.4.2) are propagated as described before, and thus they are accessible (once their physical locality is known) also to other nodes in the same subnet.

2.5 Processes & Node Coordinators

Processes are the basic computational units. The class `KlavaProcess` is an abstract class that must be specialized to create processes. The derived classes must implement the method `executeProcess()` declared as follows:

abstract public void executeProcess() **throws** KlavaException

This method will be invoked when a process is executed (just like `run` for threads). A process must be executed within a node, which will be its *execution environment*. `KlavaProcess` also offers all the methods to access tuple spaces; these methods transparently call the corresponding methods of the class `KlavaNode`.

Besides standard process a new category of processes, *node coordinators*, is introduced that, apart from the standard operations, can also perform the privileged actions dealing with node connectivity implemented by the methods presented in Section 2.4.2. For these privileged processes KLAVA provides the class `KlavaNodeCoordinator`. As required by the model, due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. Of course, in order to guarantee better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`.

Since processes and node coordinators have to forward the KLAIM actions to the node they are currently executing on, they need to store a reference to such node. In order to force the distinction between standard processes (that can only perform a limited number of actions) and node coordinators (that can also execute privileged actions), the reference is indeed a reference to a *proxy* that has a reduced interface when it is stored inside a standard process: a `KlavaNodeProcessProxy` is stored inside a standard process, while a "more powerful" `KlavaNodeCoordinatorProxy` is stored in a node coordinator. Notice that since a proxy, and not a direct reference

to the actual node, is stored, the process cannot call a method that is not allowed to call, since the proxy itself does not provide it: misuses, thanks to the different interfaces of proxies, are caught at compile time. For this reason processes and node coordinator should not be started directly but only with an **eval**: this will take care of setting the proxy; otherwise the process/node coordinator would not be able to execute any KLAVA operation.

In the previous version of KLAVA, accordingly to the original KLAIM model, when evaluating tuples, allocation environments were also used to "close" processes exchanged in communications. Indeed, evaluating a process that occurs in a field of a tuple meant substituting it with its *closure*, namely the process along with the environment of the node where the evaluation is taking place. Hence, a remarkable difference between **out**(P, l) and **eval**(P, l) was also that **out** was adding the closure of P to the tuple space located at l, while **eval** would send only P, not its closure, for execution at l. This affected the evaluation of logical localities: when a process needs to translate a logical locality into a physical one, first its own allocation environment is used (if it has one) and then, if the translation fails, the environment of the node where the process runs is used. This means that a process delivered with an **out** used a *static scoping* strategy for logical localities while a process remotely spawned with an **eval** used a *dynamic scoping* strategy. Thus, for instance, in the former case, `self` refers to the originating site, while, in the latter case, `self` refers to the current execution site.

This automatic treatment of locality translation and process closure made programming a little bit harder when dynamic scoping was needed (for instance in the scenario of the example presented in Section 4.2). For this reason this automatic mechanism was removed in the hierarchical KLAIM model (see [Bettini *et al.*, 2002a]), and we supply a finer-grain control on this: the translation of a logical locality and the closure of a process has to be explicitly obtained via, respectively, the methods `getPhysical()` and `closeProcess()`.

A process (resp., a node coordinator) can be given a name at constructor time or with the `setName()`; if no name is specified, an automatically built name (containing the name of the class of the process and an incremental number) will be given to the process (resp., node coordinator). Please keep in mind that, in case of a migrating process, the name will not be transmitted together with the process, which, upon arrival at remote site, will be given a fresh name (using the same automatic name strategy of the Java class `Thread`).

Example 2.5 (`AcceptNodeCoordinator`)**.** Since it is quite usual to have a separate thread (in this case node coordinator) waiting for connection request, KLAVA already provides a class for this (here we show only the relevant parts of the code):

```
01: package klava.topology;
02:
03: import klava.KlavaException;
04: import klava.PhysicalLocality;
05:
06: public class AcceptNodeCoordinator extends KlavaNodeCoordinator {
07:   /**
08:    * The locality to accept connections. If null, uses the main locality of
09:    * the node.
10:    */
11:   PhysicalLocality physicalLocality = null;
12:
13:   /**
14:    * The locality of the remote accepted node. Notice that each time the
15:    * accept is performed this will be overwritten, so if this coordinator is
16:    * used in loop mode, keep this in mind.
17:    */
18:   PhysicalLocality remote;
19:
20:   /**
21:    * Continuously wait for incoming connections
22:    */
23:   boolean loop = false;
24:
25:   public void executeProcess() throws KlavaException {
```

```
26:       while (true) {
27:         remote = new PhysicalLocality();
28:         boolean success = false;
29:
30:         if (physicalLocality == null) {
31:           /* use the node's main locality */
32:           success = accept(remote);
33:         } else {
34:           success = accept(physicalLocality, remote);
35:         }
36:
37:         if (success) {
38:           success(remote);
39:         }
40:
41:         if (!loop)
42:           break;
43:       }
44:
45:     }
46:
47:     /**
48:      * Called if the accept succeeded. Default is empty, subclasses should
49:      * specialize this method.
50:      *
51:      * @param remote
52:      *          The accepted locality
53:      */
54:     protected void success(PhysicalLocality remote) {
55:
56:     }
57: }
58:
```

loop → AcceptNodeCoordinator.java:23, page 13
physicalLocality → AcceptNodeCoordinator.java:11, page 13
remote → AcceptNodeCoordinator.java:18, page 13
success → AcceptNodeCoordinator.java:54, page 14

Similarly, it also provide `RegisterNodeCoordinator` that instead of performing `accept()` it performs `register()`.

□

Example 2.6 (`InOutProcess`)**.** This is a simple process that given a template tuple, and two localities, `inDestination` and `outDestination` (they both default to `self`) tries to retrieve a tuple matching the template from the the first locality and puts the so obtained tuple to the second locality.

```
01: package klava.examples.process;
02:
03: import klava.KlavaException;
04: import klava.Locality;
05: import klava.Tuple;
06: import klava.topology.KlavaProcess;
07:
08: public class InOutProcess extends KlavaProcess {
09:   /** The destination locality of in (default: self) */
10:   public Locality inDestination = self;
11:
12:   /** The destination Locality of out (default: self) */
13:   public Locality outDestination = self;
14:
15:   /** The tuple to in from the inDestination and then to out to the outDestination. */
16:   protected Tuple template;
17:
18:   public void executeProcess() throws KlavaException {
19:     SystemOutPrint("performing in" + template + "@" + inDestination + "\n");
20:     in(template, inDestination);
```

```
21:       SystemOutPrint("retrieved " + template + "\n");
22:       SystemOutPrint("performing out" + template + "@" + outDestination + "\n");
23:       out(template, outDestination);
24:       SystemOutPrint("exiting" + "\n");
25:     }
26: }
27:
```

in → TupleSpace.java:2, page 6
inDestination → InOutProcess.java:10, page 14
out → TupleSpace.java:4, page 6
outDestination → InOutProcess.java:13, page 14
template → InOutProcess.java:16, page 14

The methods `SystemOutPrint()` simply prints the string on the standard out by prefixing the process name (similarly `SystemErrPrint()` on the stadard error).

□

2.6 Some examples

Here we provide some small examples showing usage of nodes, processes and node coordinators.

Example 2.7 (ConnectedWithNewloc)**.** This example connects two nodes with `newloc()`. Then we execute on the server an instance of `InOutProcess` (Example 2.6) that will retrieve a tuple from the server (we manually insert such a matching tuple in the server) and puts the result to the client (infact we wait for the result at the client node). Remember that the localities of `InOutProcess` default to `self`, thus, since we add this process to the server, it will try to retrieve the tuple from the server.

```
01: package klava.examples.topology;
02:
03: import org.mikado.imc.common.IMCException;
04:
05: import klava.KInteger;
06: import klava.KString;
07: import klava.KlavaException;
08: import klava.PhysicalLocality;
09: import klava.Tuple;
10: import klava.examples.process.InOutProcess;
11: import klava.topology.KlavaNode;
12:
13: public class ConnectedWithNewloc {
14:
15:   public static void main(String[] args) throws IMCException, KlavaException {
16:     KlavaNode serverNode = new KlavaNode();
17:
18:     /* insert a tuple in the server node */
19:     Tuple tuple = new Tuple(new KString("foo"), new KInteger(10));
20:     serverNode.out(tuple);
21:
22:     KlavaNode clientNode = new KlavaNode();
23:
24:     /* this logs the client to the server and returns the locality of the client */
25:     PhysicalLocality clientLoc = serverNode.newloc(clientNode);
26:
27:     /* the process will look for the tuple at the server and send the result to the client */
28:     InOutProcess inOutProcess = new InOutProcess(new Tuple(new KString(), new KInteger()));
29:     inOutProcess.outDestination = clientLoc;
30:     serverNode.addNodeProcess(inOutProcess);
31:
32:     /* let's wait for the response at the client */
33:     Tuple result = new Tuple(new KString(), new KInteger());
34:     clientNode.in(result);
35:
36:     System.out.println("result: " + result);
37:     System.exit(0);
```

```
38:   }
39: }
40:
```

InOutProcess → InOutProcess.java:8, page 14
in → TupleSpace.java:2, page 6
out → TupleSpace.java:4, page 6
outDestination → InOutProcess.java:13, page 14

The output of this examples should be something similar to the following one (since the physical locality of the newloc is a fresh one, and in particular a local pipe, the result might be different each time):

```
klava.examples.process.InOutProcess-3: performing in( !KString, !KInteger )@self
klava.examples.process.InOutProcess-3: retrieved ( foo, 10 )
klava.examples.process.InOutProcess-3: performing out( foo, 10 )@pipe-/1/2
result: ( foo, 10 )
klava.examples.process.InOutProcess-3: exiting
```

□

Example 2.8 (`Net`)**.** KLAVA also provides a node class that always accepts connection requests (listening on specific physical localities). This is accomplished by using the `AcceptNodeCoordinator` shown in Example 2.5:

```
01: package klava.topology;
02:
03: import klava.KlavaException;
04: import klava.PhysicalLocality;
05:
06: import org.mikado.imc.common.IMCException;
07:
08: public class Net extends KlavaNode {
09:   /** The localities where we listen for incoming accept request */
10:   protected Vector<PhysicalLocality> localities;
11:
12:   public Net(Vector<PhysicalLocality> localities) throws IMCException {
13:     this.localities = localities;
14:     startAccept();
15:   }
16:
17:   protected void startAccept() throws IMCException {
18:     Enumeration<PhysicalLocality> locs = localities.elements();
19:     while (locs.hasMoreElements()) {
20:      AcceptNodeCoordinator acceptNodeCoordinator = new AcceptNodeCoordinator(locs.nextElement());
21:       acceptNodeCoordinator.setLoop(true);
22:       addNodeCoordinator(acceptNodeCoordinator);
23:     }
24:   }
25:
26:   public static void main(String[] args) throws KlavaException, IMCException {
27:     Vector<PhysicalLocality> localities = new Vector<PhysicalLocality>();
28:
29:     if (args.length == 0) {
30:       PhysicalLocality physicalLocality = new PhysicalLocality("localhost:9999");
31:       System.err.println("syntax: locality [localities...]");
32:       System.err.println("using default: " + physicalLocality);
33:       localities.addElement(physicalLocality);
34:     }
35:
36:     for (int i = 0; i < args.length; ++i) {
37:       localities.addElement(new PhysicalLocality(args[i]));
38:     }
39:
40:     new Net(localities);
41:   }
42: }
43:
```

AcceptNodeCoordinator → AcceptNodeCoordinator.java:6, page 13

Net → Net.java:12, page 16
Net → Net.java:8, page 16
localities → Net.java:10, page 16
physicalLocality → AcceptNodeCoordinator.java:11, page 13
startAccept → Net.java:17, page 16

Similarly, KLAVA provides the node class `LogicalNet` that uses a `RegisterNodeCoordinator`. □

Example 2.9 (`ClientNode`)**.** The following node is thought to be used as a client of another node:

```
01: package klava.topology;
02:
03: import klava.KlavaException;
04: import klava.LogicalLocality;
05: import klava.PhysicalLocality;
06:
07: public class ClientNode extends KlavaNode {
08:   /**
09:    * @param server
10:    *          The locality of the server to connect to
11:    * @throws KlavaException
12:    */
13:   public ClientNode(PhysicalLocality server) throws KlavaException {
14:     if (!login(server))
15:       throw new KlavaException("failed to log to " + server);
16:
17:     System.out.println("logged to " + server);
18:   }
19:
20:   public ClientNode(PhysicalLocality server, LogicalLocality logicalLocality)
21:       throws KlavaException {
22:     if (!subscribe(server, logicalLocality))
23:       throw new KlavaException("failed to subscribe to " + server
24:           + " as " + logicalLocality);
25:
26:     System.out.println("subscribed to " + server + " as " + logicalLocality);
27:   }
28:
29:   public static void main(String[] args) throws KlavaException {
30:     /* the default one */
31:     PhysicalLocality server = new PhysicalLocality("localhost", 9999);
32:
33:     if (args.length == 0) {
34:       System.out.println("syntax: <locality of the server> [logical locality]");
35:       System.out.println("using default: " + server);
36:     } else if (args.length > 2) {
37:       System.out.println("syntax: <locality of the server> [logical locality]");
38:       System.exit(1);
39:     } else if (args.length > 0) {
40:       server = new PhysicalLocality(args[0]);
41:     }
42:
43:     if (args.length == 2) {
44:       new ClientNode(server, new LogicalLocality(args[1]));
45:     } else {
46:       new ClientNode(server);
47:     }
48:   }
49: }
50:
```

ClientNode → ClientNode.java:13, page 17
ClientNode → ClientNode.java:20, page 17
ClientNode → ClientNode.java:7, page 17

Notice that this client can be used both for a login and for a subscribe (provided a logical locality is specified).

Now we can run the `Net` of Example 2.8 passing some physical localities where it will accept connection requests, such as:

```
java klava.topology.Net tcp-localhost:9999 udp-localhost:9999
```

Now you can run two client nodes, specifying one of these physical localities (actually you can run as many client nodes you want for each of these two physical localities) on separate terminals:

```
java klava.topology.ClientNode udp-localhost:9999

java klava.topology.ClientNode tcp-localhost:9999
```

Notice how we employ two different communication mechanisms: TCP and UDP (see Section 2.3. □

Example 2.10 (ConnectedWithLogin)**.** This example is similar to Example 2.7, but this time the two nodes are connected through TCP (see `Net` in Example 2.8 and `ClientNode` in Example 2.9). Notice that since we use the loopback address, 127.0.0.1, you can still run this program in local, without network connectivity. Then we execute on the client an instance of `InOutProcess` (Example 2.6) that will retrieve a tuple from the server (we manually insert such a matching tuple in the server) and puts the result to the client (infact we wait for the result at the client node). Remember that the localities of `InOutProcess` default to `self`, thus, since we add this process to the client, it will return the tuple to the client.

```
01: import klava.examples.process.InOutProcess;
02: import klava.topology.ClientNode;
03: import klava.topology.KlavaNode;
04: import klava.topology.Net;
05:
06: public class ConnectedWithLogin {
07:   public static void main(String[] args) throws IMCException, KlavaException {
08:     PhysicalLocality serverLoc = new PhysicalLocality("tcp-127.0.0.1:9999");
09:     KlavaNode serverNode = new Net(serverLoc);
10:
11:     /* insert a tuple in the server node */
12:     Tuple tuple = new Tuple(new KString("foo"), new KInteger(10));
13:     serverNode.out(tuple);
14:
15:     /* will automatically log to the server */
16:     KlavaNode clientNode = new ClientNode(serverLoc);
17:
18:     /* the process will look for the tuple at the server and send the result to the client */
19:     InOutProcess inOutProcess = new InOutProcess(new Tuple(new KString(), new KInteger()));
20:     inOutProcess.inDestination = serverLoc;
21:     clientNode.addNodeProcess(inOutProcess);
22:
23:     /* let's wait for the response at the client */
24:     Tuple result = new Tuple(new KString(), new KInteger());
25:     clientNode.in(result);
26:
27:     System.out.println("result: " + result);
28:     System.exit(0);
29:   }
30: }
31:
```

ClientNode → ClientNode.java:13, page 17
ClientNode → ClientNode.java:20, page 17
ClientNode → ClientNode.java:7, page 17
InOutProcess → InOutProcess.java:8, page 14
Net → Net.java:12, page 16
Net → Net.java:8, page 16
in → TupleSpace.java:2, page 6
inDestination → InOutProcess.java:10, page 14
out → TupleSpace.java:4, page 6

The output of this examples should be something similar to the following one (since the physical locality consists of an IP and a fixed port number , the result should be the same each time):

```
klava.examples.process.InOutProcess-3: performing in( !KString, !KInteger )@tcp-127.0.0.1:9999
klava.examples.process.InOutProcess-3: retrieved ( foo, 10 )
klava.examples.process.InOutProcess-3: performing out( foo, 10 )@self
klava.examples.process.InOutProcess-3: exiting
result: ( foo, 10 )
```

□

2.7 Code mobility in KLAVA

In KLAVA, processes can be sent as part of a message and executed at the destination site, where however their Java classes, i.e., their code, may be unknown. KLAVA completely relies on the IMC code mobility functionalities (described in [Bettini, 2004]); here we describe the features that are relevant to the programmer.

It might then be necessary to make such code available for execution at remote hosts; this can be done basically in two different ways:

- *automatic* approach: the classes needed by a process are collected and delivered together with the process;
- *on-demand* approach: when a Java class is needed by the remote computer that received a process for execution, it is requested to the server that did send the process.

We follow the automatic approach because it complies better with the mobile agent paradigm: during a migration, an agent takes with it all the information that it may need for later executions. The drawback of this approach is that code that may never be used by the mobile agent or that is already provided by the remote site is also shipped. However, our choice has the advantage of simplifying the handling of *disconnected operations* [Park & Reichl, 1998]: the agent owner does not have to stay connected after sending the agent and can connect later just to check whether his agent has terminated. This may not be possible with the on-demand approach: the server that sent the process must always be on-line in order to provide the classes needed by remote hosts.

Therefore, a process must be sent along with its class binary code, and with the class code of all the objects the process uses. Obviously, only the code of user defined classes has to be sent, as the other code (e.g., Java and `klava` classes) is common to every KLAVA application. This guarantees that classes belonging to `java` sub-packages are not loaded from other sources (especially, the network); this would be very dangerous, since, in general, such classes have many more access privileges.

The names of user defined classes can be retrieved by means of class introspection (*Java Reflection API*). Just before dispatching a process to a remote site, a recursive procedure is called for collecting all classes that are used by the process when declaring: data members, objects returned by or passed to a method/constructor, exceptions thrown by methods, inner classes, the interfaces implemented by its class, the base class of its class.

When extending `KlavaProcess`, there is an important detail to know in order to avoid run-time errors that would take place at remote sites and would be very hard to discover: Java Reflection API is unable to inspect local variables of methods. This implies that if a process uses a class only to declare a variable in a method, this class will not be collected and thus, when the process executes that method on a remote site, a `ClassNotFoundException` may be thrown. This limitation is due to the specific implementation of Java Reflection API, but it can be easily dealt with, once the programmer is aware of the problem.

According to the requirements made on the run-time support, code mobility may also be classified as follows [Cugola *et al.*, 1997; Hohlfeld & Yee, 1998]:

- *weak mobility*: code coming from a different site can be dynamically linked;

- *strong mobility*: a thread can move its code and execution state to a different site and resume its execution on arrival;
- *full mobility*: in addition to strong mobility, the whole state of the running program is moved, and this includes all threads' stacks, namespaces (e.g., I/O descriptors, file-system names) and other resources, so that migration is completely transparent.

Full mobility can be considered orthogonal to mobile agents and requires a strong support from the operating system layer. Strong mobility is the notion of mobility that best fits in with the classical concept of mobile agent: the execution state of a migrating agent is suspended, and its stack and program counter are sent to the destination site, together with the relevant data; at the destination site, the stack of the agent is reconstructed and the program counter is set appropriately, i.e., to the first instruction after the migration action. Instead, weak mobility does not meet the intuitive idea of mobile agent, because automatic resumption of execution thread is one of the main features of mobile agents (it exalts their autonomy).

As for the kind of mobility supplied by our framework, KLAVA only provides *weak mobility*. In general, all those systems based on Java, implement only weak mobility; this is due to the fact that Java does not permit dynamic inspection of the byte code stack and this makes impossible to save the execution state for later use. For this reason, also KLAVA can only supply *weak mobility* of agents. X-KLAIM supports strong mobility via a preprocessing performed by the compiler [Bettini & De Nicola, 2001].

Apart from the **eval** operation already discussed before, a process can also use the method `migrate()` that makes the process migrate to another site, and interrupts the execution of the migrating process at the local site; thus, basically, this method never returns. As noted above in Java we can implement only weak mobility, thus, upon arrival, the migrated process starts its execution from the beginning (its execution state was not saved); it is up to the programmer to implement a way to keep track of the execution state. An example of process that uses `migrate()` is in the Example 2.13.

Example 2.11 (`EvalProcess`)**.** This simple process spawns another process to the destination locality (default `self`) throuh **eval**.

```
01: package klava.examples.process;
02:
03: import klava.KlavaException;
04: import klava.Locality;
05: import klava.topology.KlavaProcess;
06:
07: public class EvalProcess extends KlavaProcess {
08:   /** The process to eval */
09:   public KlavaProcess klavaProcess;
10:
11:   /** Destination for eval (default: self) */
12:   public Locality destination = self;
13:
14:   public void executeProcess() throws KlavaException {
15:     SystemOutPrint("eval(" + klavaProcess.getName() + ")@" + destination + "\n");
16:     eval(klavaProcess, destination);
17:     SystemOutPrint("done\n");
18:   }
19: }
20:
```

destination → EvalProcess.java:12, page 20
klavaProcess → EvalProcess.java:9, page 20

□

Example 2.12 (EvalProcessAtRemote)**.** This example is a variant of Example 2.10, where the `InOutProcess` (Example 2.6) is sent directly to the remote server by `EvalProcess` (Example 2.11). Notice that this time we must set as the destination locality of `InOutProcess` the physical locality of the client. In order to retrieve such locality we translate the logical locality `self` to the physical one using `getPhysical()`.

```
01: public class EvalProcessAtRemote {
02:   public static void main(String[] args) throws IMCException, KlavaException {
03:     PhysicalLocality serverLoc = new PhysicalLocality("tcp-127.0.0.1:9999");
04:     KlavaNode serverNode = new Net(serverLoc);
05:
06:     /* insert a tuple in the server node */
07:     Tuple tuple = new Tuple(new KString("foo"), new KInteger(10));
08:     serverNode.out(tuple);
09:
10:     /* will automatically log to the server */
11:     KlavaNode clientNode = new ClientNode(serverLoc);
12:
13:     /* retrieve the physical locality of the client */
14:     PhysicalLocality clientLoc = clientNode.getPhysical(KlavaNode.self);
15:
16:     /* the process will be sent to the server by EvalProcess */
17:     InOutProcess inOutProcess = new InOutProcess(new Tuple(new KString(), new KInteger()));
18:     inOutProcess.outDestination = clientLoc;
19:     clientNode.addNodeProcess(new EvalProcess(inOutProcess, serverLoc));
20:
21:     /* let's wait for the response at the client */
22:     Tuple result = new Tuple(new KString(), new KInteger());
23:     clientNode.in(result);
24:
25:     System.out.println("result: " + result);
26:     System.exit(0);
27:   }
28: }
29:
```


ClientNode → ClientNode.java:13, page 17
ClientNode → ClientNode.java:20, page 17
ClientNode → ClientNode.java:7, page 17
EvalProcess → EvalProcess.java:7, page 20
InOutProcess → InOutProcess.java:8, page 14
Net → Net.java:12, page 16
Net → Net.java:8, page 16
in → TupleSpace.java:2, page 6
out → TupleSpace.java:4, page 6
outDestination → InOutProcess.java:13, page 14

The output of this examples should be something similar to the following one (since the physical locality consists of an IP and a fixed port number, the result should be the same each time for the server, while it might change every time for the client):

```
klava.examples.process.EvalProcess-4:
          eval(klava.examples.process.InOutProcess-3)@tcp-127.0.0.1:9999
klava.examples.process.EvalProcess-4: done
terminated klava.examples.process.EvalProcess-4
Thread-5: performing in( !KString, !KInteger )@self
Thread-5: retrieved ( foo, 10 )
Thread-5: performing out( foo, 10 )@tcp-127.0.0.1:38763
result: ( foo, 10 )
```

Notice that the `InOutProcess`, since migrated, did not keep its own name. The tcp-127.0.0.1:38763 is the physical locality of the client, which, as said above, might change each time the program is executed (since the TCP port is chosen each time by the operating system).

□

Example 2.13 (`MigratingProcess`)**.** This simple process migrates to a destination, using `migrate()`, and executes another process once arrived (via `executeProcess()`). It extends `EvalProcess` (Example 2.11) in order to reuse its fields.

```
01: package klava.examples.process;
02:
03: import klava.KlavaException;
04: import klava.Locality;
```

```
05: import klava.topology.KlavaProcess;
06:
07: public class MigratingProcess extends EvalProcess {
08:   /** Whether the process has already migrated */
09:   boolean migrated = false;
10:
11:   public void executeProcess() throws KlavaException {
12:     if (!migrated) {
13:       SystemOutPrint("migrating to " + destination + "\n");
14:       migrated = true;
15:       migrate(destination);
16:       // will never reach here
17:     } else {
18:       SystemOutPrint("executing " + klavaProcess.getName() + " @ " + destination + "\n");
19:       executeNodeProcess(klavaProcess);
20:       SystemOutPrint("done\n");
21:     }
22:   }
23: }
24:
```

EvalProcess → EvalProcess.java:7, page 20
destination → EvalProcess.java:12, page 20
klavaProcess → EvalProcess.java:9, page 20
migrated → MigratingProcess.java:9, page 22

As noted above in Java we can implement only weak mobility, thus, upon arrival, the migrated process starts its execution from the beginning (its execution state was not saved); it is up to the programmer to implement a way to keep track of the execution state. In this case, for instance, we use a boolean field that tells us whether we have already migrated.

□

Example 2.14 (MigrateProcessAtRemote)**.** This example is a variant of Example 2.12, where the `InOutProcess` (Example 2.6) is passed to an instance of `MigratingProcess` (Example 2.13) that will execute it once arrived to the server.

```
01: package klava.examples.mobility;
02:
03: public class MigrateProcessAtRemote {
04:
05:   public static void main(String[] args) throws IMCException, KlavaException {
06:     /* ... as EvalProcessAtRemote ... */
07:
08:     /* the process will be executed at the server by MigratingProcess */
09:     InOutProcess inOutProcess = new InOutProcess(new Tuple(new KString(),
10:         new KInteger()));
11:     inOutProcess.outDestination = clientLoc;
12:     clientNode.addNodeProcess(new MigratingProcess(inOutProcess, serverLoc));
13:
14:     /* ... as EvalProcessAtRemote ... */
15:   }
16: }
17:
```

InOutProcess → InOutProcess.java:8, page 14
MigratingProcess → MigratingProcess.java:7, page 22
outDestination → InOutProcess.java:13, page 14

The output of this examples should be something similar to the following one (since the physical locality consists of an IP and a fixed port number, the result should be the same each time for the server, while it might change every time for the client):

```
klava.examples.process.MigratingProcess-4: migrating to tcp-127.0.0.1:9999
Thread-5: executing Thread-6 @ tcp-127.0.0.1:9999
Thread-6: performing in( !KString, !KInteger )@self
Thread-6: retrieved ( foo, 10 )
Thread-6: performing out( foo, 10 )@tcp-127.0.0.1:45285
result: ( foo, 10 )
```

```
Thread-6: exiting
Thread-5: done
```

Notice that the `MigratingProcess`, since migrated, did not keep its own name (the same holds for `InOutProcess`).

□

3 Programming Graphical Applications

The KLAVA package provides, in the subpackage `gui`, a limited support for programming graphical applications. In particular it provides specific classes that implement specialized tuple spaces that have a graphical "semantics". It will then be possible, e.g., to access a button or a text field by using the `TupleSpace` interface. Correspondingly, specialized `KlavaNodes` are provided that embed these graphical objects. It will then be smooth, e.g., to access a text area that resides on a remote node (since it will be accessible via standard tuple space operations). Here we list them (first the specialized tuple space and then the specialized node embedding it)

- `TupleSpaceScreen`, `ScreenNode`: when a tuple is inserted in this tuple space (via **out**) its contents will be printed on a text area; in particular if the tuple consists only of one string, only that string is printed; other tuple space operations have no effect (and retrievial operations always return false).

- `TupleSpaceKeyboard`, `KeyboardNode`: a retrievial operation blocks the process until something is inserted in a text field (and ENTER is pressed); what is entered must have a simplified tuple syntax: tuple fields are separated by commas, by default a field is intended as a string otherwise an interger is specified by adding `:int`, a boolean by adding `:bool`.

  If one searches for a tuple of the shape `("getText", !s)` then a tuple `("getText", text)` is inserted in the tuple space, where text is the current text of the text field (thus it always returns something, possibly an empty string, and the operation never blocks). Notice that, according to the semantics of tuple space operations, if the search is performed via an **in** then the text field is also cleared.

  If one puts a tuple of the shape `("setText", s)` then the string s is inserted in the text field (replacing what was there before).

- `TupleSpaceList`, `ListNode`: all tuples inserted in this tuple space are represented in a graphical list object. It is also possible to perform other actions on the list (by using tuples):

  - If you put a tuple of the shape:

    ```
    ("COMMAND", "getSelectedItem")
    ```

    The tuple space will insert another tuple with the string representing an element of the list that is selected (or an empty string if none is selected) of the shape:

    ```
    ("COMMAND", "getSelectedItem", selectedstring)
    ```

    If you use `"getSelectedItems"` it will insert a tuple space containing all the selected items, instead of only one string.

  - If you put a tuple of the shape:

    ```
    ("COMMAND", "removeAll")
    ```

    then all the elements of the list will be removed.

- `TupleSpaceButton`, `ButtonNode`: This is a graphical button that, when pressed, makes available a tuple of the shape

  ```
  ("CLICKED")
  ```

thus, processes can intercept click events by waiting for such tuples.

The easiest way to learn how to use these graphical widgets is to see them in action in the following example.

Example 3.1 (`NodeWithScreen`)**.** This is a node that embeds a `ScreenNode`. Notice that it connects the screen node by using `newloc();` then it maps the logical locality `screen` to the physical locality of the screen node (returned by `newloc()`). `CloseableFrame` is an IMC class that provides a frame that, when closed via the close button, closes the embedded resource (in this case the top level node).

```
01: package klava.examples.gui;
02:
03: import org.mikado.imc.gui.CloseableFrame;
04:
05: import klava.KlavaException;
06: import klava.LogicalLocality;
07: import klava.PhysicalLocality;
08: import klava.gui.ScreenNode;
09: import klava.topology.KlavaNode;
10:
11: public class NodeWithScreen extends KlavaNode {
12:   public final static LogicalLocality screenLoc = new LogicalLocality("screen");
13:
14:   public NodeWithScreen(String nodeName) throws KlavaException {
15:     setNodeName(nodeName);
16:     ScreenNode screenNode = new ScreenNode(nodeName + "'s screen");
17:     PhysicalLocality screenPhyLoc = newloc(screenNode);
18:     addToEnvironment(screenLoc, screenPhyLoc);
19:     CloseableFrame nodeFrame = new CloseableFrame(this);
20:     nodeFrame.setTitle(nodeName);
21:     nodeFrame.add(screenNode.getPanel());
22:     nodeFrame.setVisible(true);
23:   }
24:
25:   public static void main(String args[]) throws Exception {
26:     new NodeWithScreen("screen");
27:   }
28: }
29:
```

NodeWithScreen → NodeWithScreen.java:11, page 24
NodeWithScreen → NodeWithScreen.java:14, page 24
screenLoc → NodeWithScreen.java:12, page 24
screenNode → ChatClientFrame.java:7, page 36
screenNode → GuiNodeExample.java:50, page 25

□

Example 3.2 (`GuiNodeExample`)**.** This node uses four subnodes, `ScreenNode`, `KeyboardNode`, `ListNode` and `ButtonNode` and composes them in a frame. By taking a look at the process' code it should be quite easy to understand what they do. Moreover, this example uses the `newloc()` passing an instance of `KlavaNode`. A picture of this example is in Screenshot 1.

```
001: package klava.examples.gui;
002:
003: public class GuiNodeExample extends JFrame {
004:   /**
005:    * Waits for button click events and show the selected items from the list
006:    * in the screen
007:    */
008:   public class ButtonProcess extends KlavaProcess {
009:     public void executeProcess() throws KlavaException {
010:       Tuple click = new Tuple(TupleSpaceButton.clickedString);
011:       Tuple requestTuple = new Tuple(TupleSpaceList.cmdString,
012:           new KString("getSelectedItems"));
013:
```

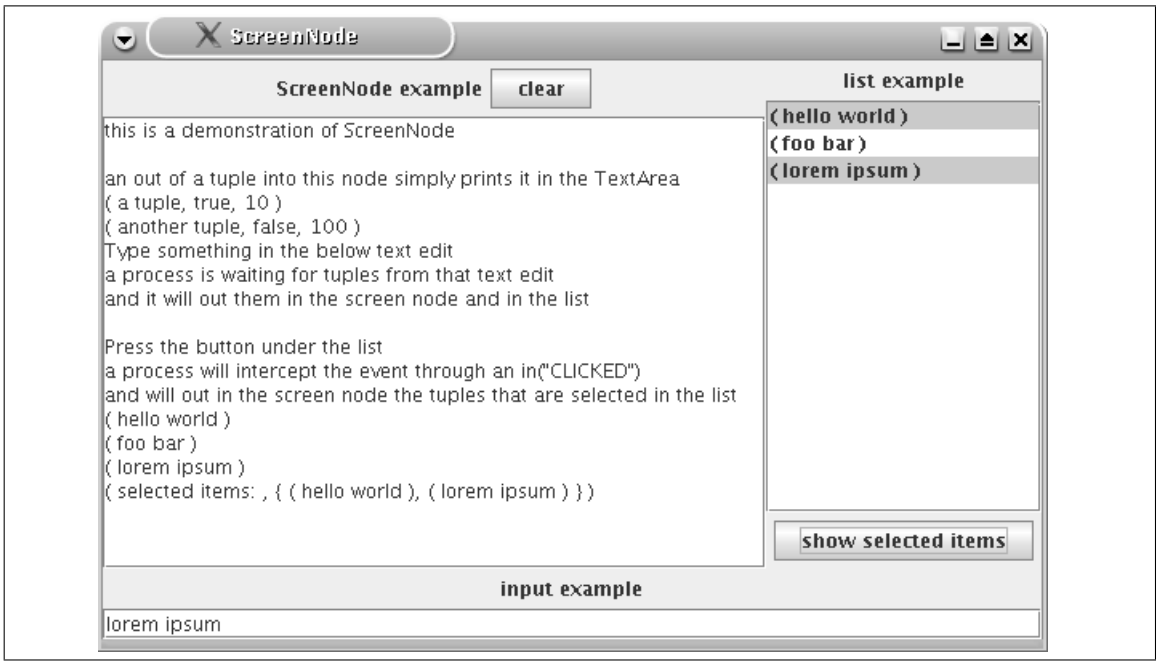

ScreenNode
ScreenNode example
clear
list example
this is a demonstration of ScreenNode
an out of a tuple into this node simply prints it in the TextArea
(a tuple, true, 10)
(another tuple, false, 100)
Type something in the below text edit
a process is waiting for tuples from that text edit
and it will out them in the screen node and in the list
Press the button under the list
a process will intercept the event through an in("CLICKED")
and will out in the screen node the tuples that are selected in the list
(hello world)
(foo bar)
(lorem ipsum)
(selected items: , { (hello world), (lorem ipsum) })
(hello world)
(foo bar)
(lorem ipsum)
show selected items
input example
lorem ipsum

Figure 1: The `GuiNodeExample`.

```
014:          while (true) {
015:            /* wait for a click event */
016:            in(click, button);
017:
018:            /* request selected items to the list */
019:            out(requestTuple, list);
020:
021:            /* retrieve the selected items in the list */
022:            TupleSpace selected = new TupleSpaceVector();
023:            Tuple selectedItems = new Tuple(TupleSpaceList.cmdString,
024:                new KString("getSelectedItems"), selected);
025:            in(selectedItems, list);
026:
027:            /* print the selected items in the screen */
028:            out(new Tuple("selected items: ", selected.toString()), screen);
029:          }
030:        }
031:    }
032:
033:    /**
034:     * Collects the tuples inserted in the input field and outs them in the
035:     * ScreenNode and in the ListNode
036:     */
037:    public class CollectorProcess extends KlavaProcess {
038:      public void executeProcess() throws KlavaException {
039:        while (true) {
040:          /* reads a string from the text field */
041:          Tuple tuple = new Tuple(new KString());
042:          in(tuple, keyboard);
043:          out(tuple, list);
044:          out(new Tuple(tuple + "\n"), screen);
045:        }
046:      }
047:    }
048:
049:    KlavaNode node = new KlavaNode();
050:    ScreenNode screenNode;
051:    KeyboardNode keyboardNode;
052:    ButtonNode buttonNode;
053:    ListNode listNode;
```

```
054:
055:    Locality screen;
056:    Locality list;
057:    Locality keyboard;
058:    Locality button;
059:
060:    public static void main(String[] args) throws KlavaException, IMCException {
061:      GuiNodeExample screenNodeExample = new GuiNodeExample();
062:      screenNodeExample.setDefaultCloseOperation(EXIT_ON_CLOSE);
063:      screenNodeExample.pack();
064:      screenNodeExample.setVisible(true);
065:    }
066:
067:    public GuiNodeExample() throws KlavaException, IMCException {
068:      screenNode = new ScreenNode("ScreenNode example");
069:      screenNode.out(new Tuple("this is a demonstration of ScreenNode\n\n"));
070:      screenNode.out(...); /* other instruction strings omitted here */
071:
072:      /* connects the ScreenNode to the main Node */
073:      screen = node.newloc(screenNode);
074:
075:      keyboardNode = new KeyboardNode("input example");
076:
077:      /* connects the KeyboardNode to the main Node */
078:      keyboard = node.newloc(keyboardNode);
079:
080:      listNode = new ListNode("list example");
081:
082:      /* connects the ListNode to the main Node */
083:      list = node.newloc(listNode);
084:
085:      node.addNodeProcess(new CollectorProcess());
086:
087:      buttonNode = new ButtonNode("show selected items");
088:
089:      button = node.newloc(buttonNode);
090:
091:      node.addNodeProcess(new ButtonProcess());
092:
093:      initialize();
094:    }
095:
096:    private void initialize() {
097:      this.setSize(300, 200);
098:      this.setContentPane(getJContentPane());
099:      this.setTitle("ScreenNode");
100:     this.add(screenNode.getPanel(), BorderLayout.CENTER);
101:     this.add(keyboardNode.getPanel(), BorderLayout.SOUTH);
102:     this.getJPanel().add(listNode.getPanel(), BorderLayout.CENTER);
103:     this.getJPanel().add(buttonNode.getPanel(), BorderLayout.SOUTH);
104:   }
105: }
106:
```


ButtonProcess → GuiNodeExample.java:8, page 24
CollectorProcess → GuiNodeExample.java:37, page 25
GuiNodeExample → GuiNodeExample.java:3, page 24
GuiNodeExample → GuiNodeExample.java:67, page 26
TupleSpace → TupleSpace.java:1, page 6
button → GuiNodeExample.java:58, page 26
buttonNode → GuiNodeExample.java:52, page 25
in → TupleSpace.java:2, page 6
initialize → GuiNodeExample.java:96, page 26
keyboard → GuiNodeExample.java:57, page 26
keyboardNode → GuiNodeExample.java:51, page 25
list → GuiNodeExample.java:56, page 26
listNode → GuiNodeExample.java:53, page 25
node → ChatClientFrame.java:5, page 36
node → GuiNodeExample.java:49, page 25
out → TupleSpace.java:4, page 6

screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
screenNode → ChatClientFrame.java:7, page 36
screenNode → GuiNodeExample.java:50, page 25

□

4 Three Example Applications

In this section we present three programming examples that rely on mobility and distribution. The first example concerns a *news gatherer* that exploits mobile agents for retrieving information on remote sites; the second example implements a *load balancing system* that dynamically redistributes mobile code among several processors; the last example is a simplified *chat system*. The main purposes of these examples, whose core implementation parts are reported in some code snippets throughout the following sections, is showing that, by using KLAVA, dealing with mobility and communications among distributed processes boils down to a few method calls, since KLAVA takes care of all the low level details of code mobility and distributed synchronization.

4.1 A news gatherer

In this section we show how to program in KLAVA a *news gatherer* that relies on mobile agents for retrieving information on remote sites. We assume that some data is distributed over the nodes of a KLAVA net (a sort of distributed database) and that each node either contains the information we are searching for, or the locality of the next node to visit in the net. A slightly different version of this scenario is implemented in X-KLAIM in [Bettini, 2003b].

This is the implementation of the `NewsGatherer`:

```
01: package klava.examples.newsgatherer;
02:
03: public class NewsGatherer extends KlavaProcess {
04:   /** The item to find in the database. */
05:   KString itemToFind;
06:
07:   /** Where to return results. */
08:   Locality homeLoc;
09:
10:   /** The locality of the screen of the home locality. */
11:   Locality homeScreen;
12:
13:   public void executeProcess() throws KlavaException {
14:     // formal field for the value of the item to search
15:     KString itemVal = new KString();
16:     // formal field for the (possible) next locality to visit
17:     Locality nextLoc = new PhysicalLocality();
18:     // the local screen
19:     LogicalLocality screen = new LogicalLocality("screen");
20:     out(new Tuple("searching for " + itemToFind + " at " +
21:             getPhysical(self) + "\n"), homeScreen);
22:
23:     if (read_nb(new Tuple(itemToFind, itemVal), self)) {
24:       out(new Tuple("found item " + itemVal + "\n"), screen);
25:
26:       // we found the item, communicate it home
27:       out(new Tuple(itemToFind, itemVal), homeLoc);
28:
29:       return; // we finished
30:     } else if (read_nb(new Tuple(itemToFind, nextLoc), self)) {
31:       // let's migrate to the next locality of the database
32:       out(new Tuple("found next locality " + nextLoc + "\n"), screen);
33:
34:       migrate(nextLoc);
```

```
35:       } else {
36:         // we really failed :-(
37:         out(new Tuple(itemToFind, "failed"), homeLoc);
38:         return;
39:       }
40:     }
41: }
42:
```

homeLoc → NewsGatherer.java:8, page 27
homeScreen → NewsGatherer.java:11, page 27
itemToFind → NewsGatherer.java:5, page 27
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26

The agent `NewsGatherer` first searches for the tuple containing the value of the desired item (using the non-blocking variant of **read**); if it finds it, it communicates it back home. Otherwise it tries to retrieve the locality of the next node of the database to visit and if it finds it it migrates there (where it will start the execution again). Notice that it uses the logical locality `screen` to refer to the current execution site's screen (that must be provided by each node visited by the agent). Moreover, it also uses the locality of the home's screen to communicate to the home site what it is doing.

The node of the client, `NewsClientNode`, provides a screen too (it derives from `NodeWithScreen`, Example 3.1) and simply subscribes to a given server locality, by using the specified logical locality:

```
1: package klava.examples.newsgatherer;
2:
3: public class NewsClientNode extends NodeWithScreen {
4:   public NewsClientNode(String nodeName, Locality serverLoc)
5:       throws KlavaException {
6:     super(nodeName);
7:     subscribe(serverLoc, new LogicalLocality(nodeName));
8:   }
9: }
10:
```

NodeWithScreen → NodeWithScreen.java:11, page 24
NodeWithScreen → NodeWithScreen.java:14, page 24

A node of the data base, `DataBaseNode` is similar (we do not show it here). Instead, we show an example of how constructing the distributed data base:

```
01: package klava.examples.newsgatherer;
02:
03: public class DatabaseExample {
04:   /**
05:    * We'll have this final net configuration:
06:    *
07:    * <pre>
08:    * main node   (links item to node1)
09:    *    |
10:    *  node 1    (links item to node2)
11:    *    |
12:    *  node 2    (links item to node3)
13:    *    |
14:    *  node 3    (contains value for item)
15:    * </pre>
16:    */
17:   protected void initDB(PhysicalLocality server) throws KlavaException,
18:       IMCException {
19:     DatabaseNode mainDatabaseNode = new DatabaseNode("main node");
20:     mainDatabaseNode.subscribe(server, new LogicalLocality("main node"));
21:
22:     DatabaseNode databaseNode3 = new DatabaseNode("node 3");
23:     DatabaseNode databaseNode2 = new DatabaseNode("node 2");
```

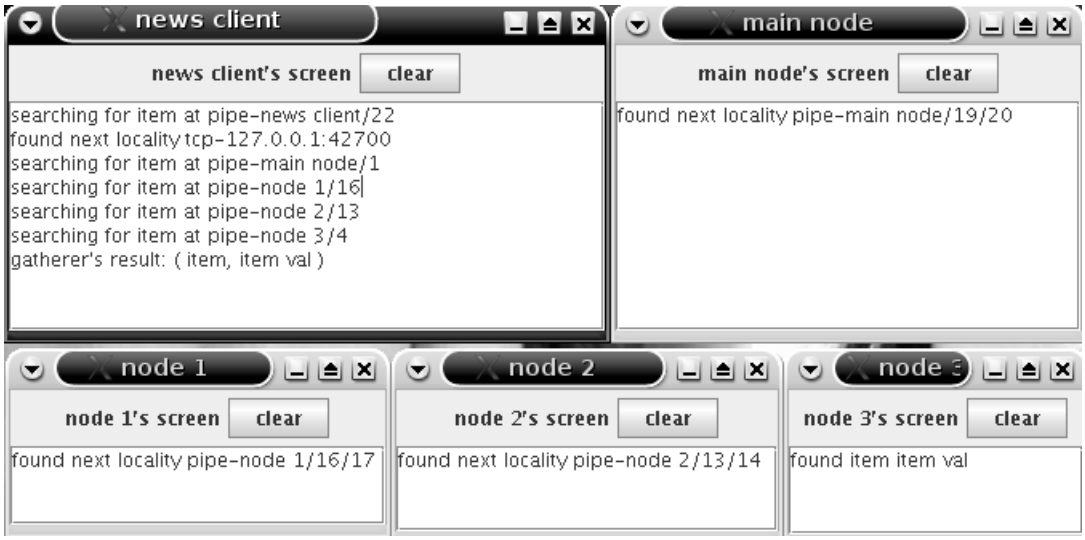
```
24:     DatabaseNode databaseNode1 = new DatabaseNode("node 1");
25:
26:     PhysicalLocality databaseNode3Loc = databaseNode2.newloc(databaseNode3);
27:     PhysicalLocality databaseNode2Loc = databaseNode1.newloc(databaseNode2);
28:     PhysicalLocality databaseNode1Loc = mainDatabaseNode.newloc(databaseNode1);
29:
30:     KString item = new KString("item");
31:
32:     mainDatabaseNode.out(new Tuple(item, databaseNode1Loc));
33:     databaseNode1.out(new Tuple(item, databaseNode2Loc));
34:     databaseNode2.out(new Tuple(item, databaseNode3Loc));
35:     databaseNode3.out(new Tuple(item, new KString("item val")));
36:   }
37: }
38:                                                    out → TupleSpace.java:4, page 6
```

This creates a distributed (hierarchical) database that consists of four nodes, each one containing the information of the next site to visit to find information about the specific item (in particular the last node contains the information).

Now we can put all together in a test application where there's a central server where both the client node and the distributed data base subscribe:

```
01: package klava.examples.newsgatherer;
02:
03: public class NewsGathererApplication {
04:   public NewsGathererApplication(PhysicalLocality serverLoc)
05:       throws KlavaException, IMCException, InterruptedException {
06:     LogicalNet newsNet = new LogicalNet(serverLoc);
07:
08:     new DatabaseExample(serverLoc);
09:
10:     NewsClientNode newsClientNode = new NewsClientNode("news client", serverLoc);
11:
12:     // this will give the initial information to the NewsGatherer
13:     KString item = new KString("item");
14:     newsClientNode.out(new Tuple(item,
15:         newsClientNode.getPhysical(new LogicalLocality("main node"))));
16:
17:     LogicalLocality clientLocScreen = new LogicalLocality("screen");
18:     LogicalLocality clientLoc = new LogicalLocality("news client");
19:     newsClientNode.addNodeProcess(new NewsGatherer(item, clientLoc,
20:         newsClientNode.getPhysical(clientLocScreen)));
21:
22:     // now wait for response from the NewsGatherer
23:     Tuple responseFromGatherer = new Tuple(item, new KString());
24:     newsClientNode.in(responseFromGatherer);
25:     newsClientNode.out(new Tuple("gatherer's result: "), clientLocScreen);
26:     newsClientNode.out(responseFromGatherer, clientLocScreen);
27:   }
28: }
29:                                  DatabaseExample → DatabaseExample.java:3, page 28
                                    NewsClientNode → NewsClientNode.java:3, page 28
                                    NewsClientNode → NewsClientNode.java:4, page 28
                                        NewsGatherer → NewsGatherer.java:3, page 27
                                                in → TupleSpace.java:2, page 6
                                               out → TupleSpace.java:4, page 6
```

This code also spawns the `NewsGatherer` at the client node, where we inserted a tuple allowing the gatherer to migrate to the first node of the data base. Then we wait for the response at the client node. Screenshot 4.1 shows the agent that visits the four nodes of the distributed data base, and the information printed on each node.



Screenshot 4.1: A news gatherer agent visiting the four nodes of the distributed data base.

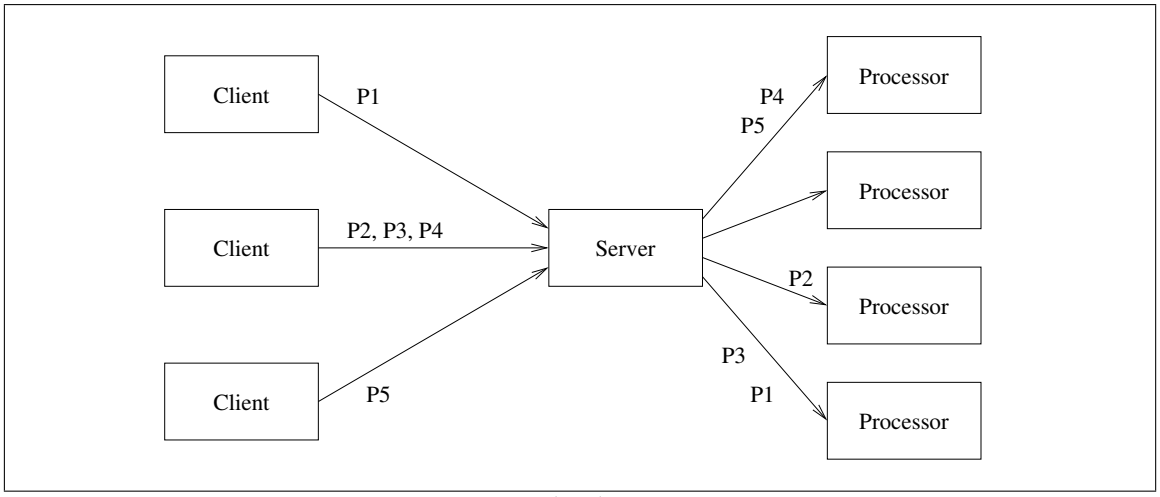


Figure 2: Load Balancing System

4.2 Load balancing

In this second scenario, we suppose that remote clients send processes for execution to a server node that distributes the received processes among a group of processors by using, each time, the (estimated) idlest one (Figure 2). This is determined by using the *Leaky Bucket of Credits* pattern [Adams *et al.*, 1996]: when entering the net managed by the load balancing server, each processor sends a number of "credits" to the server (this number corresponds to the processor availability to perform computations on behalf of the server); the server stores the number of credits in a database and, when needed, it chooses the processor with the highest number of credits and decreases this number. The server may exhaust all credits; in that case it executes the process locally.

When a processor receives a process, it immediately starts executing the process (in a parallel thread) and sends a credit back to the server (represented by the locality `processorServer`). Indeed, the Leaky Bucket Of Credits pattern is based on the heuristic that if a processor is busy, it cannot send a credit back, or at least it does not send a credit immediately. This behavior is implemented by the class `ProcessorNode`:

```
01: package klava.examples.loadbalancing;
02:
03: public class ProcessorNode extends NodeWithScreen {
```

```
04:   public class Executor extends KlavaProcess {
05:     public void executeProcess() throws KlavaException {
06:       while (true) {
07:         KlavaProcessVar klavaProcessVar = new KlavaProcessVar();
08:         /* wait for a process... */
09:         in(new Tuple(klavaProcessVar), self);
10:         out(new Tuple("executing "
11:             + klavaProcessVar.klavaProcess.getName() + "...\n"),
12:             screenLoc);
13:         /* execute it locally */
14:         eval(klavaProcessVar.klavaProcess, self);
15:         /* send a credit back */
16:         out(new Tuple("sending credit back...\n"), screenLoc);
17:         out(creditTuple, loadBalancingServerLoc);
18:       }
19:     }
20:
21:   }
22:
23:   public final static KString creditString = new KString("CREDIT");
24:
25:   /** The tuple to send a credit back */
26:   public Tuple creditTuple;
27:
28:   /** The locality of the load balancing server */
29:   Locality loadBalancingServerLoc;
30:
31:   /** The logical locality with which this node is known to the load balancing server */
32:   LogicalLocality myLogicalLocality;
33:
34:   public ProcessorNode(String nodeName, Locality serverLoc, int initialCredits)
35:       throws KlavaException {
36:     super(nodeName);
37:     myLogicalLocality = new LogicalLocality(nodeName);
38:     subscribe(serverLoc, myLogicalLocality);
39:     loadBalancingServerLoc = serverLoc;
40:     creditTuple = new Tuple(creditString, myLogicalLocality);
41:
42:     /* send the initial credits */
43:     out(new Tuple(myLogicalLocality, new KInteger(initialCredits)), serverLoc);
44:
45:     out(new Tuple("registered at " + serverLoc + " as " + nodeName + "\n"), screenLoc);
46:
47:     /* start the executor process */
48:     eval(new Executor(), self);
49:   }
50: }
51:
```


Executor → ProcessorNode.java:4, page 31
NodeWithScreen → NodeWithScreen.java:11, page 24
NodeWithScreen → NodeWithScreen.java:14, page 24
creditString → ProcessorNode.java:23, page 31
creditTuple → ProcessorNode.java:26, page 31
in → TupleSpace.java:2, page 6
klavaProcess → EvalProcess.java:9, page 20
loadBalancingServerLoc → ProcessorNode.java:29, page 31
myLogicalLocality → ProcessorNode.java:32, page 31
out → TupleSpace.java:4, page 6
screenLoc → NodeWithScreen.java:12, page 24

This node subscribes to the `LoadBalancingNode` (shown later) by also sending an initial amount of credits, and spawns a process that simply waits for a tuple containing a process (by using a `KlavaProcessVar`), spawns the received process for execution at the current site and sends a credit back to `LoadBalancingNode`.

The `LoadBalancingNode` executes some processes (whose classes are inner classes); The first one we present it a `KlavaNodeCoordinator`, `RegisterProcessor`, that takes care of registering processor nodes that subscribe and receive the initial credits. The association between the log-

ical locality of a processor node and its current number of credits are stored in the `TupleSpace` `credits` (the `lockTuple` is used to lock this "database"):

```
01: public class RegisterProcessors extends KlavaNodeCoordinator {
02:   /** Where we wait for subscription requests */
03:   PhysicalLocality registerLocality;
04:
05:   public void executeProcess() throws KlavaException {
06:     while (true) {
07:       PhysicalLocality processorPhysicalLocality = new PhysicalLocality();
08:       LogicalLocality processorLogicalLocality = new LogicalLocality();
09:       register(registerLocality, processorPhysicalLocality, processorLogicalLocality);
10:
11:       /* wait for the initial credits, within 3 seconds */
12:       KInteger initialCredits = new KInteger();
13:       if (!in_t(new Tuple(processorLogicalLocality, initialCredits), self, 3000)) {
14:         SystemErrPrint("credits not received from "
15:                 + processorPhysicalLocality + "("
16:                 + processorLogicalLocality + ")\n");
17:         return;
18:       }
19:
20:       credits.in(lockTuple);
21:       out(new Tuple("initial credits from "
22:               + processorLogicalLocality + ": " + initialCredits
23:               + "\n"), screenLoc);
24:       credits.out(new Tuple(processorLogicalLocality, initialCredits));
25:       credits.out(lockTuple);
26:     }
27:   }
28: }
29:
```

credits → LoadBalancingNode.java:11, page 34
in → TupleSpace.java:2, page 6
in_t → TupleSpace.java:7, page 7
lockTuple → LoadBalancingNode.java:14, page 34
out → TupleSpace.java:4, page 6
registerLocality → RegisterProcessor.java:3, page 32
screenLoc → NodeWithScreen.java:12, page 24

Another process, `CreditReceiver`, waits for credit that processor nodes send back and updates the database:

```
01: public class CreditReceiver extends KlavaProcess {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       LogicalLocality processorLocality = new LogicalLocality();
05:       in(new Tuple(ProcessorNode.creditString, processorLocality), self);
06:
07:       /* lock the database for updating */
08:       credits.in(lockTuple);
09:
10:       KInteger currentCredits = new KInteger(); // formal
11:       if (credits.in_nb(new Tuple(processorLocality, currentCredits))) {
12:         /* update the number of credits */
13:         credits.out(new Tuple(processorLocality, new KInteger(currentCredits.intValue() + 1)));
14:         out(new Tuple("updated credits for " + processorLocality + "\n"), screenLoc);
15:       } else {
16:         /* better error handling */
17:         out(new Tuple("unknown processor " + processorLocality + "\n"), screenLoc);
18:       }
19:
20:       /* release the database */
21:       credits.out(lockTuple);
22:     }
23:   }
24: }
25:
```

ProcessorNode → ProcessorNode.java:3, page 30

ProcessorNode → ProcessorNode.java:34, page 31
creditString → ProcessorNode.java:23, page 31
credits → LoadBalancingNode.java:11, page 34
in → TupleSpace.java:2, page 6
in_nb → TupleSpace.java:10, page 7
lockTuple → LoadBalancingNode.java:14, page 34
out → TupleSpace.java:4, page 6
screenLoc → NodeWithScreen.java:12, page 24

Finally, the process `AcceptProcesses` waits for a process, sends it to the processor node with the current highest credit number (or executes it locally if all credits are 0) and updates the credit database. Notice that it makes use of iteration capabilities over a tuple space with `resetOriginalTemplate()`:

```
01: public class AcceptProcesses extends KlavaProcess {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       KlavaProcessVar klavaProcessVar = new KlavaProcessVar();
05:       in(new Tuple(klavaProcessVar), self);
06:       out(new Tuple("received process " + klavaProcessVar.klavaProcess.getName() + "\n"),
07:         screenLoc);
08:
09:       /* lock the database */
10:       credits.in(lockTuple);
11:       LogicalLocality processorLoc = new LogicalLocality(); // formal
12:       KInteger creditNum = new KInteger(); // formal
13:       LogicalLocality idlestProcessor = null;
14:       int maxCredits = 0;
15:       Tuple creditTuple = new Tuple(processorLoc, creditNum);
16:       while (credits.read_nb(creditTuple)) {
17:         if (maxCredits < creditNum.intValue()) {
18:           /* found another candidate */
19:           maxCredits = creditNum.intValue();
20:           idlestProcessor = new LogicalLocality(processorLoc);
21:         }
22:         /* reset the template for iteration */
23:         creditTuple.resetOriginalTemplate();
24:       }
25:       /* if we found an idle processor */
26:       if (idlestProcessor != null) {
27:         out(new Tuple("executing process "
28:                 + klavaProcessVar.klavaProcess.getName()
29:                 + " at " + idlestProcessor + "\n"), screenLoc);
30:         out(new Tuple(klavaProcessVar.klavaProcess), idlestProcessor);
31:       } else {
32:         /* execute it locally */
33:         out(new Tuple("executing process "
34:                 + klavaProcessVar.klavaProcess.getName()
35:                 + " locally\n"), screenLoc);
36:         eval(klavaProcessVar.klavaProcess, self);
37:       }
38:       /* update the credits of the processor in the database */
39:       credits.in(new Tuple(idlestProcessor, new KInteger(maxCredits)));
40:       credits.out(new Tuple(idlestProcessor, new KInteger(maxCredits - 1)));
41:       /* release lock on the database */
42:       credits.out(lockTuple);
43:     }
44:   }
45: }
46:
```

creditTuple → ProcessorNode.java:26, page 31
credits → LoadBalancingNode.java:11, page 34
in → TupleSpace.java:2, page 6
klavaProcess → EvalProcess.java:9, page 20
lockTuple → LoadBalancingNode.java:14, page 34
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7

screenLoc → NodeWithScreen.java:12, page 24

The class `LoadBalancingNode` consists of these three inner classes and this initialization code:

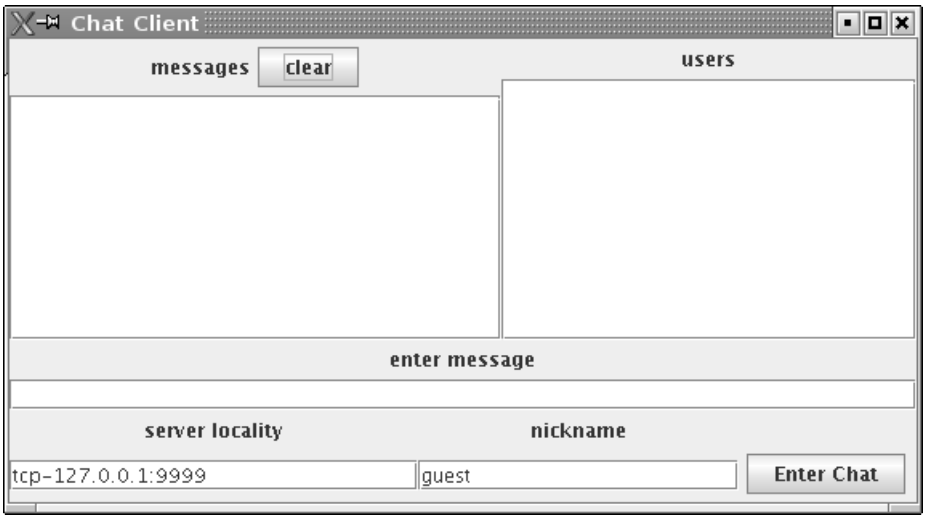
```
01: package klava.examples.loadbalancing;
02:
03: public class LoadBalancingNode extends NodeWithScreen {
04:   /** INNER CLASSES SHOWN ABOVE */
05:
06:   /**
07:    * This will store the associations (tuples) of the shape (processor
08:    * locality, num of credits). An additional tuple, ("LOCK") will be used to
09:    * guarantee synchronization when iterating over this tuple space.
10:    */
11:   TupleSpace credits = new TupleSpaceVector();
12:
13:   /** The tuple used to lock the credits database */
14:   Tuple lockTuple = new Tuple("LOCK");
15:
16:   public LoadBalancingNode(String nodeName, PhysicalLocality acceptLoc,
17:       Locality serverLoc) throws KlavaException, IMCException {
18:     super(nodeName);
19:     /* since we always use this very same tuple */
20:     lockTuple.setHandleRetrieved(false);
21:     credits.out(lockTuple); // initialize the credit database
22:     addNodeCoordinator(new RegisterProcessors(acceptLoc));
23:     eval(new AcceptProcesses());
24:     eval(new CreditReceiver());
25:
26:     if (serverLoc != null)
27:       subscribe(serverLoc, new LogicalLocality(nodeName));
28:   }
29: }
30:
```

AcceptProcesses → AcceptProcesses.java:1, page 33
CreditReceiver → CreditReceiver.java:1, page 32
NodeWithScreen → NodeWithScreen.java:11, page 24
NodeWithScreen → NodeWithScreen.java:14, page 24
RegisterProcessors → RegisterProcessor.java:1, page 32
TupleSpace → TupleSpace.java:1, page 6
credits → LoadBalancingNode.java:11, page 34
lockTuple → LoadBalancingNode.java:14, page 34
out → TupleSpace.java:4, page 6

Notice that since the tuple `lockTuple` is removed and inserted in the tuple space `credits` it is crucial that `setHandleRetrieved()` is used (as examplained in Section 2.1).

The following code runs a load balancing system where there are 4 processor nodes, and many processes (whose code is not shown here, but their intent is to perform some busy loops) on the load balancing node:

```
01: package klava.examples.loadbalancing;
02:
03: public class LoadBalancingExample {
04:   public static void main(String[] args)
05:     throws KlavaMalformedPhyLocalityException, KlavaException, IMCException {
06:     PhysicalLocality loadBancingServerLoc = new PhysicalLocality("tcp-127.0.0.1:9999");
07:     LoadBalancingNode loadBalancingNode = new LoadBalancingNode(
08:         "load balancing node", loadBancingServerLoc, null);
09:
10:     new ProcessorNode("processor 1", loadBancingServerLoc, 2);
11:     new ProcessorNode("processor 2", loadBancingServerLoc, 2);
12:     new ProcessorNode("processor 3", loadBancingServerLoc, 3);
13:     new ProcessorNode("processor 4", loadBancingServerLoc, 1);
14:
15:     /* now spawn some processes on the load balancing system */
16:     for (int i = 0; i < 30; ++i)
```



Screenshot 4.2: The `ChatClient` frame.

```
17:         loadBalancingNode.out(new Tuple(new BusyLoopProcess()));
18:     }
19: }
20:
```

LoadBalancingNode → LoadBalancingNode.java:16, page 34
LoadBalancingNode → LoadBalancingNode.java:3, page 34
ProcessorNode → ProcessorNode.java:3, page 30
ProcessorNode → ProcessorNode.java:34, page 31
out → TupleSpace.java:4, page 6

Let us observe that a process has to be delivered to the server by means of an **out**, since **eval** would automatically starts the execution of the process at the destination node, and thus there would be no easy way of redirecting such process to a specific processor. In the previous versions of KLAVA the closure of such process would have been actually delivered, due to the automatic evaluation mechanism, and thus, the process would have not been able to access the actual executing site by means of `self` (that would be bounded to the starting client site). This shows that the removal of this automatic evaluation mechanism in the new version of KLAVA makes programming these kinds of application much easier.

4.3 A chat system

The chat system we present in this section is simplified, but it implements the basic features that are present in several chat systems. Though this example does not deal with mobile code, it shows how to use KLAVA to implement distributed applications that can communicate through distributed and located tuple spaces. An X-KLAIM chat system similar to the one shown here is presented in [Bettini, 2003b].

The system consists of a `ChatServer` and many `ChatClients`. A client that wants to enter the chat must subscribe at the chat server. The server must keep track of all the registered clients and, when a client sends a message, the server has to deliver the message to every connected client. If the message is a private one, it will be delivered only to the clients in the list specified along with the message.

First of all, let's take a look at the graphical interface of a `ChatClient` (the interface of a `ChatServer` is similar, so we omit it here), depicted in Screenshot 4.2. All the elements of this frame are tuple spaces and nodes specialized for GUI applications, as already seen in previous examples, and explained in Section 3. When the client presses the button "Enter chat" the chat client will try to subscribe to the chat server whose locality is specified in the text field "server locality" by using the logical locality specified in the text field "nickname" (thus, the logical locality used for subscription corresponds to the nickname that the client will use in the chat). In particular, after a

successful subscription, the button will change the label to "Leave chat" that the client can use to leave the chat. At each moment, the client will see the clients currently in the chat (in the "users" list). The client can enter a message that will be sent to the chat by using the text field "enter message", and the messages of the chat will be displayed in the text area "messages". When sending a message, the user can select specific users in the list, and the message will be sent only to the selected users (private message).

Although we skip all the parts related to Java GUI programming details (e.g., inserting the panels in the right positions, see Example 3.2), we show how to connect the GUI nodes, and in particular, how to bind their physical localities to some logical localities, that will be used in the following, to refer to the elements of the frame (text fields, lists, etc.). This is a snippet of the frame of a chat client, `ChatClientFrame`:

```
01: package klava.examples.chat;
02:
03: public class ChatClientFrame extends CloseableFrame {
04:   /** The node implementing the ChatClient */
05:   KlavaNode node = new KlavaNode();
06:
07:   public ScreenNode screenNode;
08:
09:   /** To interact with the chat server */
10:   public KeyboardNode serverKeyboardNode;
11:
12:   /** To specify the nick for the chat server */
13:   public KeyboardNode nickKeyboardNode;
14:
15:   /** To enter chat messages */
16:   public KeyboardNode messageKeyboardNode;
17:
18:   /** To enter */
19:   public ButtonNode serverButtonNode;
20:
21:   public ListNode usersListNode;
22:
23:   public static final LogicalLocality screen = new LogicalLocality("screen");
24:   public static final LogicalLocality usersList = new LogicalLocality("users");
25:   public static final LogicalLocality serverKeyboard = new LogicalLocality("serverKeyboard");
26:   public static final LogicalLocality nickKeyboard = new LogicalLocality("nickKeyboard");
27:   public static final LogicalLocality messageKeyboard = new LogicalLocality("messageKeyboard");
28:   public static final LogicalLocality serverButton = new LogicalLocality("serverButton");
29:
30:   public ChatClientFrame() throws KlavaException {
31:     /* when we close the frame we close the node */
32:     setCloseable(node);
33:
34:     /* initialize and connect GUI nodes */
35:     messageKeyboardNode = new KeyboardNode("enter message");
36:     node.addToEnvironment(messageKeyboard, node.newloc(messageKeyboardNode));
37:
38:     serverKeyboardNode = new KeyboardNode("server locality");
39:     node.addToEnvironment(serverKeyboard, node.newloc(serverKeyboardNode));
40:
41:     nickKeyboardNode = new KeyboardNode("nickname");
42:     node.addToEnvironment(nickKeyboard, node.newloc(nickKeyboardNode));
43:
44:     serverButtonNode = new ButtonNode("Enter Chat");
45:     node.addToEnvironment(serverButton, node.newloc(serverButtonNode));
46:
47:     screenNode = new ScreenNode("messages");
48:     node.addToEnvironment(screen, node.newloc(screenNode));
49:
50:     usersListNode = new ListNode("users");
51:     node.addToEnvironment(usersList, node.newloc(usersListNode));
52:
53:     initialize();
54:   }
```

```
55: }
56:
```


initialize → GuiNodeExample.java:96, page 26
messageKeyboard → ChatClientFrame.java:27, page 36
messageKeyboardNode → ChatClientFrame.java:16, page 36
nickKeyboard → ChatClientFrame.java:26, page 36
nickKeyboardNode → ChatClientFrame.java:13, page 36
node → ChatClientFrame.java:5, page 36
node → GuiNodeExample.java:49, page 25
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
screenNode → ChatClientFrame.java:7, page 36
screenNode → GuiNodeExample.java:50, page 25
serverButton → ChatClientFrame.java:28, page 36
serverButtonNode → ChatClientFrame.java:19, page 36
serverKeyboard → ChatClientFrame.java:25, page 36
serverKeyboardNode → ChatClientFrame.java:10, page 36
usersList → ChatClientFrame.java:24, page 36
usersListNode → ChatClientFrame.java:21, page 36

Now let's start to see the most important parts of `ChatClient` class; this uses a `ChatClientFrame` and spawns some processes and node coordinators in the node of the `ChatClientFrame`. The following one is the node coordinator, `ChatSubscribeCoordinator`, that takes care of subscribing to the chat server, when the user presses the "Enter chat" button:

```
01: public class ChatSubscribeCoordinator extends KlavaNodeCoordinator {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       /* wait for the button to be pressed */
05:       in(new Tuple(TupleSpaceButton.clickedString), serverButton);
06:
07:       KString serverLocString = new KString();
08:       KString nickName = new KString();
09:
10:       try {
11:         if (serverPhysicalLocality == null) {
12:           /* check that the user specified all the required info */
13:           if (!(read_nb(new Tuple(TupleSpaceKeyboard.getTextString, serverLocString),
14:                   serverKeyboard)
15:              && read_nb(new Tuple(TupleSpaceKeyboard.getTextString, nickName), nickKeyboard)
16:              && serverLocString.length() > 0 && nickName.length() > 0)) {
17:             out(new Tuple("you must specify server locality and nickname\n"), screen);
18:             continue;
19:           }
20:
21:           serverPhysicalLocality = new PhysicalLocality(serverLocString);
22:           myNickName = new LogicalLocality(nickName);
23:
24:           out(new Tuple("entering chat...\n"), screen);
25:           if (!subscribe(serverPhysicalLocality, myNickName)) {
26:             out(new Tuple("entering chat failed\n"), screen);
27:             continue;
28:           }
29:
30:           out(new Tuple("entered chat " + serverPhysicalLocality + "\n"), screen);
31:
32:           /* change button label */
33:           out(new Tuple("Leave Chat"), serverButton);
34:
35:           /* get the current clients (timeout: 5 seconds) */
36:           TupleSpaceVector currentClients = new TupleSpaceVector();
37:           if (in_t(new Tuple(serverString, currentClients), self, 5000)) {
38:             LogicalLocality clientLoc = new LogicalLocality();
39:             Tuple clientTuple = new Tuple(clientLoc);
40:             while (currentClients.read_nb(clientTuple)) {
41:               out(new Tuple(new LogicalLocality(clientLoc)), usersList);
42:               clientTuple.resetOriginalTemplate();
43:             }
```

```
44:             }
45:
46:             eval(new ChatDisconnectedCoordinator());
47:           } else {
48:             unsubscribe(serverPhysicalLocality, myNickName);
49:             out(new Tuple("left chat " + serverPhysicalLocality + "\n"), screen);
50:
51:             /* reset button label */
52:             out(new Tuple("Enter Chat"), serverButton);
53:
54:             /* clear client list */
55:             out(new Tuple(TupleSpaceList.cmdString, TupleSpaceList.removeAllString), usersList);
56:
57:             serverPhysicalLocality = null;
58:           }
59:         } catch (KlavaException e) {
60:           new ExceptionMessageBox(null, e).setVisible(true);
61:         }
62:       }
63:     }
64: }
```

ChatDisconnectedCoordinator → ChatDisconnectedCoordinator.java:1, page 38
in → TupleSpace.java:2, page 6
in_t → TupleSpace.java:7, page 7
length → TupleSpace.java:12, page 7
nickKeyboard → ChatClientFrame.java:26, page 36
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
serverButton → ChatClientFrame.java:28, page 36
serverKeyboard → ChatClientFrame.java:25, page 36
usersList → ChatClientFrame.java:24, page 36

This node coordinator, once the user presses the button, checks whether the client is already in the chat (the physical locality of the server is not null); if so, it interprets the button click as "leave the chat" and thus, uses the `unsubscribe()` to leave the chat (it also clears the user list, and resets the button label back to "Enter Chat"). Otherwise, it tries to subscribe to the chat using the information present in the text fields (notice that it uses a tuple with the string `"getText"`, i.e., `TupleSpaceKeyboard.getTextString`, to read the current text in the text fields). It then, waits for the list of the current users in the chat (with a timeout of 5 seconds). Then, it spawns another node coordinator, `ChatDisconnectedCoordinator`, that will intercept a disconnection from the chat server, using `disconnected()`:

```
01: public class ChatDisconnectedCoordinator extends KlavaNodeCoordinator {
02:   public void executeProcess() throws KlavaException {
03:     PhysicalLocality disconnectedPhyLoc = new PhysicalLocality();
04:
05:     disconnected(disconnectedPhyLoc);
06:
07:     out(new Tuple("disconnected from chat " + disconnectedPhyLoc + "\n"), screen);
08:
09:     /* clear client list */
10:     out(new Tuple(TupleSpaceList.cmdString, TupleSpaceList.removeAllString), usersList);
11:
12:     serverPhysicalLocality = null;
13:
14:     /* reset button label */
15:     out(new Tuple("Enter Chat"), serverButton);
16:   }
17: }
```

out → TupleSpace.java:4, page 6
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
serverButton → ChatClientFrame.java:28, page 36
usersList → ChatClientFrame.java:24, page 36

The following process waits for the user to enter a message (and press ENTER) and then sends the message to the chat server. Notice that, before actually sending the message, the process checks whether there are users selected in the user list, since this would mean that the message is a private one (destined only for the selected users):

```
01: public class ChatMessageSender extends KlavaProcess {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       KString messageBody = new KString();
05:       in(new Tuple(messageBody), messageKeyboard);
06:
07:       /* check whether some users are selected in the list */
08:       TupleSpaceVector selectedRecipients = new TupleSpaceVector();
09:       out(new Tuple(TupleSpaceList.cmdString, TupleSpaceList.getSelectedItemsString), usersList);
10:       in(new Tuple(TupleSpaceList.cmdString, TupleSpaceList.getSelectedItemsString,
11:               selectedRecipients), usersList);
12:
13:       /* we must convert strings to LogicalLocalities */
14:       KString clientName = new KString();
15:       Tuple clientNames = new Tuple(clientName);
16:       while (selectedRecipients.in_nb(clientNames)) {
17:         selectedRecipients.out(new Tuple(new LogicalLocality(clientName)));
18:         clientNames.resetOriginalTemplate();
19:       }
20:
21:       out(new Tuple(messageString, messageBody, myNickName,
22:               selectedRecipients), serverPhysicalLocality);
23:     }
24:   }
25: }
```

in → TupleSpace.java:2, page 6
in_nb → TupleSpace.java:10, page 7
messageKeyboard → ChatClientFrame.java:27, page 36
out → TupleSpace.java:4, page 6
usersList → ChatClientFrame.java:24, page 36

In particular, a message is sent to the server as a tuple of the shape

```
("MSG", <message body>, <logical locality of the sender>, <list of recipients>)
```

If the list of recipients (implemented through a tuple space) is empty, then the message is not private and will forwarded to everyone.

Then, we have two processes that receive messages from the chat server, and show them on the screen; in particular, the first one receives standard messages, while the second one receives messages concerning users entering/leaving the chat (in that case it also updates the list of current users):

```
01: public class ChatMessageReceiver extends KlavaProcess {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       KString messageBody = new KString();
05:       LogicalLocality sender = new LogicalLocality();
06:       KBoolean privateMessage = new KBoolean();
07:
08:       in(new Tuple(messageString, messageBody, sender, privateMessage), self);
09:       out(new Tuple((privateMessage.booleanValue() ? "PRIV " : "")
10:               + "(" + sender + "): " + messageBody + "\n"), screen);
11:     }
12:   }
13: }
14:
15: public class ChatServerMessageReceiver extends KlavaProcess {
16:   public void executeProcess() throws KlavaException {
17:     while (true) {
18:       KString messageBody = new KString();
19:       LogicalLocality client = new LogicalLocality();
```

```
20:
21:         in(new Tuple(serverString, messageBody, client), self);
22:         out(new Tuple("SERVER: " + client + "  " + messageBody + "  chat\n"), screen);
23:         if (messageBody.equals(enteredString)) {
24:           if (!read_nb(new Tuple(client), usersList)) {
25:             out(new Tuple(client), usersList);
26:           }
27:         } else { // it's a LEFT message
28:           in_nb(new Tuple(client), usersList);
29:         }
30:       }
31:     }
32: }
```

in → TupleSpace.java:2, page 6
in_nb → TupleSpace.java:10, page 7
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
usersList → ChatClientFrame.java:24, page 36

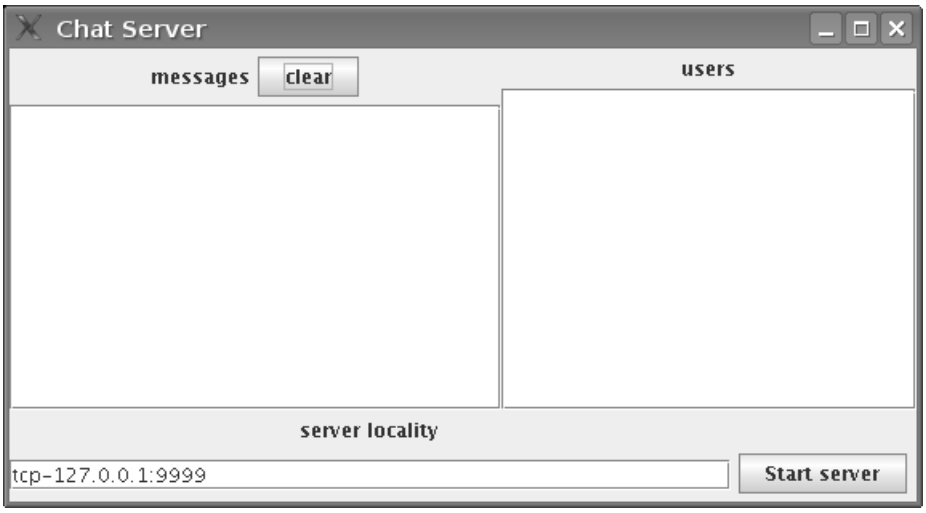
Finally, the initialization of the `ChatClient` is as follows (we omit further details of the class):

```
01: package klava.examples.chat;
02:
03: public class ChatClient {
04:   public ChatClient(String serverLoc, String nick) throws KlavaException, IMCException {
05:     chatClientFrame = new ChatClientFrame();
06:
07:     KlavaNode node = chatClientFrame.getNode();
08:
09:     /* initialize the GUI */
10:     node.out(new Tuple(new KString("setText"), serverLoc), serverKeyboard);
11:     node.out(new Tuple(new KString("setText"), nick), nickKeyboard);
12:
13:     node.eval(new ChatMessageSender());
14:     node.eval(new ChatMessageReceiver());
15:     node.eval(new ChatServerMessageReceiver());
16:     node.addNodeCoordinator(new ChatSubscribeCoordinator());
17:
18:     chatClientFrame.setVisible(true);
19:   }
20:
21:   public static void main(String[] args) throws KlavaException, IMCException {
22:     new ChatClient("tcp-127.0.0.1:9999", "guest");
23:   }
24: }
```

ChatClient → ChatClient.java:3, page 40
ChatClient → ChatClient.java:4, page 40
ChatClientFrame → ChatClientFrame.java:3, page 36
ChatClientFrame → ChatClientFrame.java:30, page 36
ChatMessageReceiver → ChatMessageReceiver.java:1, page 39
ChatMessageSender → ChatMessageSender.java:1, page 39
ChatServerMessageReceiver → ChatMessageReceiver.java:15, page 39
ChatSubscribeCoordinator → ChatSubscribeCoordinator.java:1, page 37
nickKeyboard → ChatClientFrame.java:26, page 36
node → ChatClientFrame.java:5, page 36
node → GuiNodeExample.java:49, page 25
out → TupleSpace.java:4, page 6
serverKeyboard → ChatClientFrame.java:25, page 36

The `ChatServerFrame`, Screenshot 4.3, is similar to `ChatClientFrame`, and its code is omitted here. The user of the server decides on which locality the server is waiting for client subscriptions, and when to start/stop the server, using the text field and the button. This is taken care of by the following node coordinator:

```
01: public class ChatStartStopAcceptCoordinator extends KlavaNodeCoordinator {
02:   boolean accepting = false;
```



Screenshot 4.3: The `ChatServer` frame.

```
03:
04:   public void executeProcess() throws KlavaException {
05:     while (true) {
06:       in(new Tuple(TupleSpaceButton.clickedString), serverButton);
07:
08:       if (accepting) {
09:         closeSessions(serverPhysicalLocality);
10:         accepting = false;
11:         out(new Tuple("Start server"), serverButton);
12:         out(new Tuple("server stopped\n"), screen);
13:       } else {
14:         KString serverLoc = new KString();
15:
16:         if (!read_nb(new Tuple(TupleSpaceKeyboard.getTextString, serverLoc), serverKeyboard)
17:           || serverLoc.length() == 0) {
18:           out(new Tuple("unspecified server locality\n"), screen);
19:           continue;
20:         }
21:
22:         try {
23:           serverPhysicalLocality = new PhysicalLocality(serverLoc);
24:         } catch (KlavaMalformedPhyLocalityException e) {
25:           new ExceptionMessageBox(null, e).setVisible(true);
26:         }
27:
28:         chatRegisterCoordinator = new ChatRegisterCoordinator();
29:         eval(chatRegisterCoordinator);
30:         accepting = true;
31:         out(new Tuple("Stop server"), serverButton);
32:         out(new Tuple("server started\n"), screen);
33:       }
34:     }
35:   }
36: }
```

ChatRegisterCoordinator → ChatRegisterCoordinator.java:1, page 42
accepting → ChatRegisterCoordinator.java:2, page 42
accepting → ChatStartStopCoordinator.java:2, page 40
in → TupleSpace.java:2, page 6
length → TupleSpace.java:12, page 7
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
serverButton → ChatClientFrame.java:28, page 36
serverKeyboard → ChatClientFrame.java:25, page 36

When the user stops the server, we use the method `closeSessions()` to close all the sessions (connections) that involve the locality of the server, which corresponds to close connection with all the clients. Moreover, this will also stops (via an exception) the node coordinator that registers new client subscriptions:

```
01: public class ChatRegisterCoordinator extends KlavaNodeCoordinator {
02:   boolean accepting = false;
03:
04:   public void executeProcess() throws KlavaException {
05:     while (true) {
06:       out(new Tuple("accepting clients...\n"), screen);
07:
08:       PhysicalLocality clientPhysicalLocality = new PhysicalLocality();
09:       LogicalLocality clientNick = new LogicalLocality();
10:
11:       if (register(serverPhysicalLocality, clientPhysicalLocality, clientNick)) {
12:         out(new Tuple(clientNick + " entered chat\n"), screen);
13:         out(new Tuple(clientNick), usersList);
14:
15:         /* now send the list of current clients */
16:         TupleSpaceVector currentClients = new TupleSpaceVector();
17:         LogicalLocality clientLoc = new LogicalLocality();
18:         Tuple clientTuple = new Tuple(clientLoc);
19:         while (read_nb(clientTuple, usersList)) {
20:           currentClients.out(new Tuple(clientLoc));
21:
22:           if (!(clientNick.equals(clientLoc))) {
23:             /* notify other clients that someone entered the chat */
24:             eval(new ChatServerMessageDeliver(enteredString, new LogicalLocality(clientLoc),
25:                                               clientNick));
26:             /* make a copy of the clientLoc, since it will be reset by resetOriginalTemplate */
27:           }
28:
29:           clientTuple.resetOriginalTemplate();
30:         }
31:         out(new Tuple(serverString, currentClients), clientPhysicalLocality);
32:       }
33:     }
34:   }
35: }
```

out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
usersList → ChatClientFrame.java:24, page 36

This coordinator, when a client enters the chat, sends the new client the list of current users, and notifies all the other clients that a new client entered the chat. Notice that it does not send messages to clients directly, but it spawns a concurrent process for each client, `ChatServerMessageDeliver` (not shown here, since it simply sends the message to a client via an out operation). This way the server can handle further messages, and client requests, and it does not risk to get stuck due to a client not responding. Another node coordinator, detects clients leaving the chat, via the `disconnected();` this is similar to the corresponding coordinator in the client that we have already seen, so we do not show it here. Of course, when a client leaves the chat, the server notifies all the other clients about this event.

The following process receives a message from a client and forwards it to all the other clients; if the list (tuple space) of recipients is not empty, then the message is a private one, and thus the message is sent only to the clients specified in the list:

```
01: public class ChatMessageDispatcher extends KlavaProcess {
02:   public void executeProcess() throws KlavaException {
03:     while (true) {
04:       KString messageBody = new KString();
05:       TupleSpace specificRecipients = new TupleSpaceVector();
06:       LogicalLocality sender = new LogicalLocality();
```

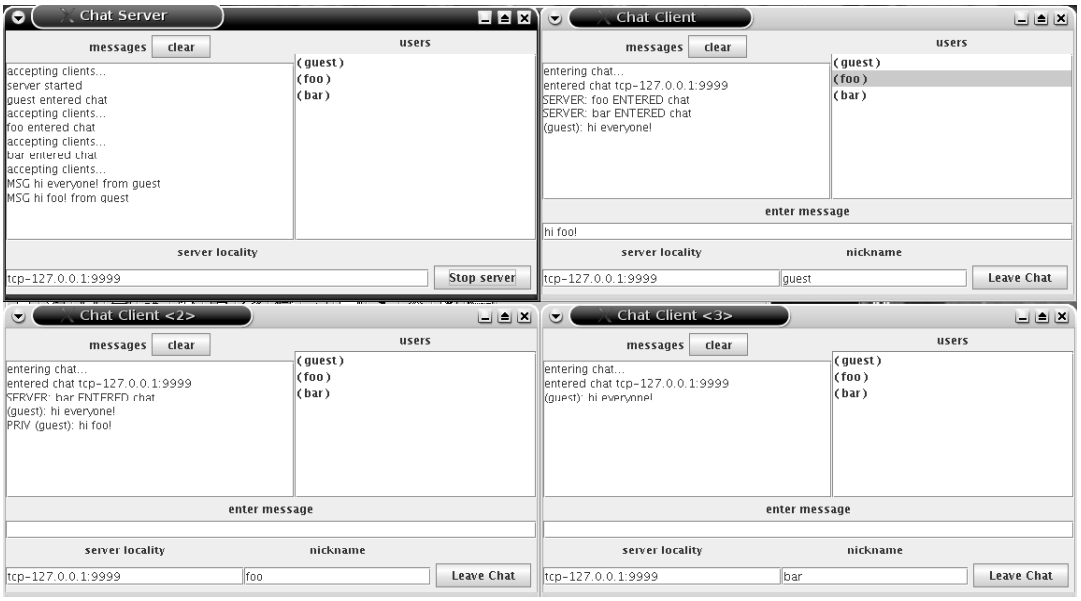
```
07:
08:         in(new Tuple(messageString, messageBody, sender, specificRecipients), self);
09:
10:         out(new Tuple("MSG " + messageBody + " from " + sender + "\n"), screen);
11:
12:         if (specificRecipients.length() == 0) {
13:           /* send to everyone */
14:           Tuple recipient = new Tuple(new LogicalLocality());
15:           while (read_nb(recipient, usersList)) {
16:             /* make a copy of the logical locality... */
17:             eval(new ChatMessageDeliver(messageBody,
18:                                 new LogicalLocality(recipient.getItem(0)
19:                                                 .toString()), sender, new KBoolean(
20:                                                                       false)), self);
21:             /* ...since this will reset recipient.getItem(0) */
22:             recipient.resetOriginalTemplate();
23:           }
24:         } else {
25:           /* send to the specified recipients */
26:           Tuple recipient = new Tuple(new LogicalLocality());
27:           while (specificRecipients.read_nb(recipient)) {
28:             eval(new ChatMessageDeliver(messageBody, new LogicalLocality(recipient.getItem(0)
29:                                                     .toString()), sender,
30:                                 new KBoolean(true)), self);
31:             recipient.resetOriginalTemplate();
32:           }
33:         }
34:       }
35:     }
36: }
```

TupleSpace → TupleSpace.java:1, page 6
in → TupleSpace.java:2, page 6
length → TupleSpace.java:12, page 7
out → TupleSpace.java:4, page 6
read_nb → TupleSpace.java:9, page 7
screen → ChatClientFrame.java:23, page 36
screen → GuiNodeExample.java:55, page 26
usersList → ChatClientFrame.java:24, page 36

Finally, the initialization of the `ChatServer` is as follows:

```
01: public class ChatServer {
02:   public ChatServer() throws KlavaException, IMCException {
03:     chatServerFrame = new ChatServerFrame();
04:
05:     KlavaNode node = chatServerFrame.getNode();
06:
07:     /* initialize the GUI */
08:     node.out(new Tuple(new KString("setText"), "tcp-127.0.0.1:9999"), serverKeyboard);
09:
10:     node.eval(new ChatMessageDispatcher());
11:     node.addNodeCoordinator(new ChatUnregisterCoordinator());
12:     node.addNodeCoordinator(new ChatStartStopAcceptCoordinator());
13:
14:     chatServerFrame.setVisible(true);
15:   }
16:
17:   public static void main(String[] args) throws KlavaException, IMCException {
18:     new ChatServer();
19:   }
20: }
```

ChatMessageDispatcher → ChatMessageDispatcher.java:1, page 42
ChatServer → ChatServer.java:1, page 43
ChatServer → ChatServer.java:2, page 43
ChatStartStopAcceptCoordinator → ChatStartStopCoordinator.java:1, page 40
node → ChatClientFrame.java:5, page 36
node → GuiNodeExample.java:49, page 25
out → TupleSpace.java:4, page 6
serverKeyboard → ChatClientFrame.java:25, page 36



Screenshot 4.4: The chat example.

Screenshot 4.4 shows the server and three clients in the chat. Notice that the user "guest" also sends a private message to "foo" (indeed, "bar" does not receive that message).

References

ADAMS, M., COPLIEN, J., GAMOKE, R., HANMER, R., KEEVE, F., & NICODEMUS, K. 1996. Fault-tolerant telecommunication system patterns. *Pages 549–562 of:* VLISSIDES, J.M., & COPLIEN, J.O. (eds), *Pattern Languages of Program Design 2*. Addison-Wesley.

BETTINI, L. 1998 (April). *Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile*. Master thesis, Dip. di Sistemi e Informatica, Univ. di Firenze.

BETTINI, L. 2003a. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. Ph.D. thesis, Dip. di Matematica, Università di Siena. Available at `http://music.dsi.unifi.it`.

BETTINI, L. 2003b. X-KLAIM*: a Programming Language for Object-Oriented Mobile Code. User's manual*. 1 edn. Dip. di Sistemi e Informatica, Univ. di Firenze. Available at `http://music.dsi.unifi.it/xklaim`.

BETTINI, L. 2004. A Java Package for Transparent Code Mobility. *Pages 112–122 of:* GUELFI, N., REGGIO, G., & ROMANOVSKY, A. (eds), *FIDJI 2004, Int. Workshop on scientific engineering of distributed Java applications*. LNCS, vol. 3409. Springer.

BETTINI, L., & DE NICOLA, R. 2001. Translating Strong Mobility into Weak Mobility. *Pages 182–197 of:* PICCO, G. P. (ed), *Mobile Agents*. LNCS, no. 2240. Springer.

BETTINI, L., & DE NICOLA, R. 2005. Mobile Distributed Programming in X-KLAIM. *Pages 29–68 of:* BERNARDO, M., & BOGLIOLO, A. (eds), *Formal Methods for Mobile Computing, Advanced Lectures*. LNCS, vol. 3465. Springer.

BETTINI, L., LORETI, M., & PUGLIESE, R. 2002a. An Infrastructure Language for Open Nets. *Pages 373–377 of: Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*. ACM.

BETTINI, L., DE NICOLA, R., & PUGLIESE, R. 2002b. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, **32**(14), 1365–1394.

BETTINI, L., BONO, V., DE NICOLA, R., FERRARI, G., GORLA, D., LORETI, M., MOGGI, E., PUGLIESE, R., TUOSTO, E., & VENNERI, B. 2003. The KLAIM Project: Theory and Practice. *Pages 88–150 of:* PRIAMI, C. (ed), *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Revised Papers*. LNCS, vol. 2874. Springer.

BETTINI, L., DE NICOLA, R., FALASSI, D., LACOSTE, M., & LORETI, M. 2005. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. *Pages 181–193 of: Proc. of 5th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*. LNCS, vol. 3543. Springer.

CARRIERO, N., & GELERNTER, D. 1989a. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, **21**(3), 323–357.

CARRIERO, N., & GELERNTER, D. 1989b. Linda in Context. *Comm. of the ACM*, **32**(4), 444–458.

CUGOLA, G., GHEZZI, C., PICCO, G.P., & VIGNA, G. 1997. Analyzing Mobile Code Languages. *In:* VITEK, J., & TSCHUDIN, C. (eds), *Mobile Object Systems*. LNCS, no. 1222. Springer.

DE NICOLA, R., FERRARI, G., & PUGLIESE, R. 1998. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, **24**(5), 315–330.

DEUGO, D. 2001. Choosing a Mobile Agent Messaging Model. *Pages 278–286 of: Proc. of ISADS 2001*. IEEE.

GELERNTER, D. 1985. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, **7**(1), 80–112.

GELERNTER, D. 1989. Multiple Tuple Spaces in Linda. *Pages 20–27 of:* ODIJK, E., REM, M., & SYRE, J. (eds), *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89)*. LNCS, vol. 365. Springer.

HOHLFELD, M., & YEE, B.S. 1998. *How to Migrate Agents*. Available at `http://www.cs.ucsd.edu/~bsy`.

PARK, A.S., & REICHL, P. 1998. Personal Disconnected Operations with Mobile Agents. *In: Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*.